AMSTRAD
464/664/6128

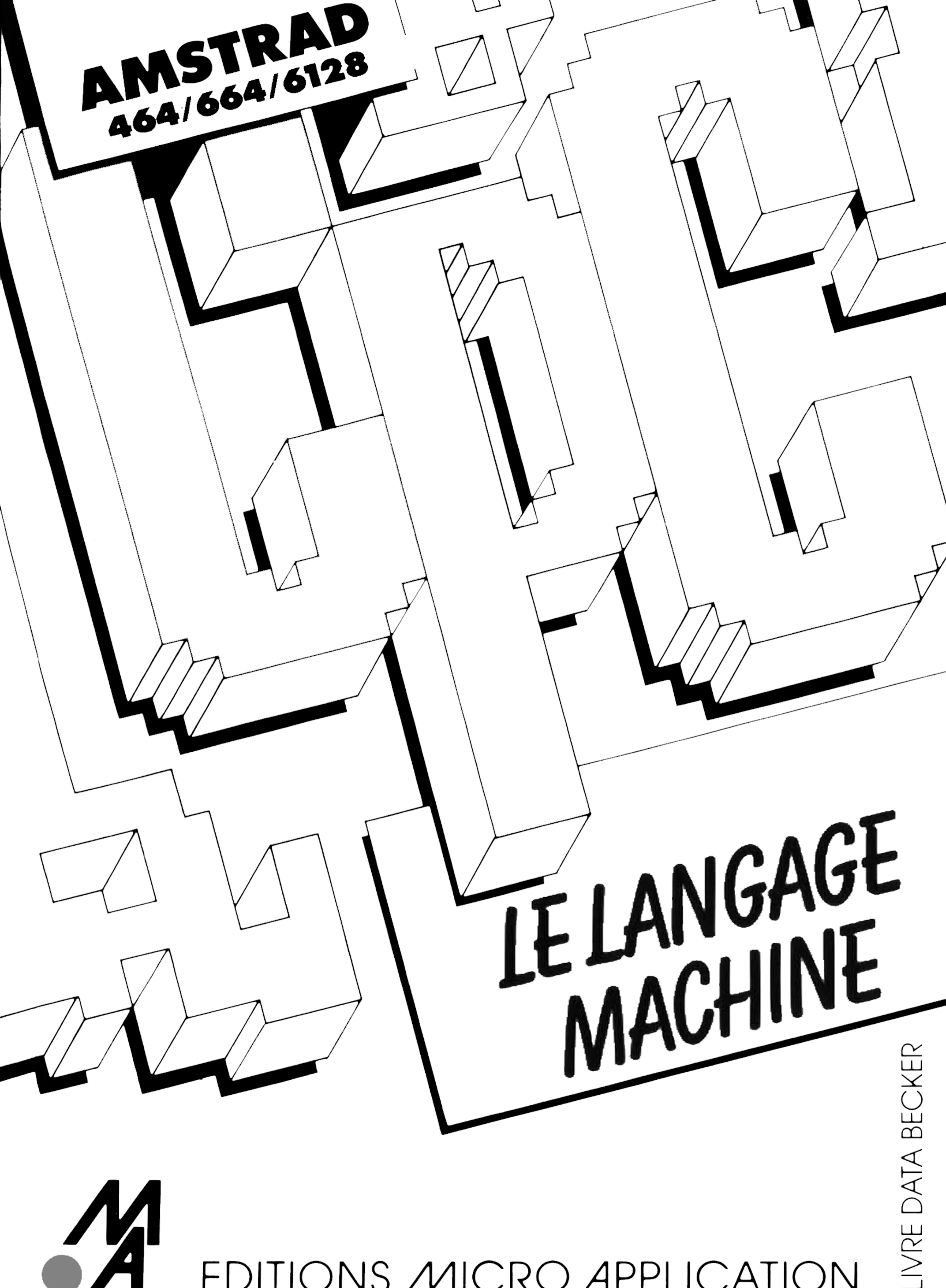
CPC

LE LANGAGE MACHINE

LIVRE DATA BECKER



EDITIONS MICRO APPLICATION


AMSTRAD
464/664/6128
LE LANGAGE MACHINE
LIVRE DATA BECKER
EDITIONS MICRO APPLICATION

Distribué par : MICRO APPLICATION
13, Rue Sainte Cécile
75009 PARIS

et

EDITIONS RADIO
189, Rue Saint Jacques
75005 PARIS

ISBN : 2-86899-015-0

Code : 282

Traduction Française assurée par Pascal HAUSSMAN

Collection dirigée par Mr Philippe OLIVIER
Edition réalisée par Frédérique BEAUDONNET

SOMMAIRE

4. Les instructions

1. Introduction

1.1. QU'EST-CE QUE LE LANGAGE-MACHINE ?

Le langage-machine est le langage de programmation que l'ordinateur peut traiter directement. Comment faut-il l'entendre ? Comme vous le savez déjà certainement, chaque ordinateur possède un micro-processeur que l'on peut qualifier de "cerveau" de l'ordinateur. Ce circuit intégré est appelé unité centrale ou CPU (Central Processing Unit). L'unité centrale exécute les instructions machine, commande la marche de l'ordinateur et les appareils connectés ou périphériques. L'unité centrale est le composant le plus important d'un ordinateur. Lorsque nous programmons en langage-machine, nous utilisons des instructions qui appellent directement l'unité centrale et que celle-ci peut exécuter directement. Le langage-machine dépend donc par conséquent de chaque type de processeur. L'AMSTRAD CPC possède un processeur Z80A qui est également utilisé dans de nombreux autres micro-ordinateurs. Le Z80A est une unité centrale très puissante qui comprend plus de 600 instructions qui sont traitées très rapidement sur le CPC.

Pourquoi donc le langage-machine ?

La plupart des ordinateurs familiaux sont dotés du Basic. Comme vous l'avez certainement déjà remarqué, ce langage n'est pas difficile à apprendre. Le Basic de l'Amstrad se distingue particulièrement par son grand nombre d'instructions. On pourrait donc penser qu'un tel Basic est parfaitement satisfaisant et qu'il est à même de bien résoudre tous les problèmes de programmation.
Pour comprendre où résident les avantages du langage-machine, il nous faut d'abord savoir comment l'ordinateur traite le Basic.

Essayez de vous représenter :

> Le ministre des affaires étrangères Basic négocie avec son collègue M. CPU au pays du langage-machine. Malheureusement, ses connaissances dans ce langage sont très réduites et il

est obligé d'utiliser les services de Mme Interpréteur qui traduit ses phrases en langage-machine. Comme vous le pensez bien, Mme Interpréteur, bien qu'étant une excellente interprète finit toujours après l'homme politique. Il en résulte un ralentissement inutile de la négociation.

C'est le même problème qui se pose pour la programmation en Basic. L'ordinateur doit tout d'abord faire interpréter par l'interpréteur le programme écrit en Basic par le programmeur. L'interpréteur Basic constitue une partie du système d'exploitation. Il interprète le programme instruction par instruction. Il procède ensuite immédiatement à l'exécution de l'instruction. Plus précisément : l'interpréteur reconnaît l'instruction Basic puis déclenche l'exécution de l'instruction Basic en appelant la routine machine correspondant à une instruction donnée.

Par exemple :

MODE 2

L'interpréteur lit cette instruction caractère par caractère. Les *espaces, les double-points et les virgules lui indiquent qu'un mot* est terminé. L'interpréteur compare ce mot (MODE) avec les entrées figurant dans la table des instructions dans le système d'exploitation. S'il ne trouve pas ce mot, il essaie de l'interpréter comme étant le nom d'une variable. Si cela ne marche pas non plus, un message d'erreur est sorti. *Si l'interpréteur trouve le mot, il* saute à l'adresse de saut affectée à ce mot. C'est là que la valeur suivante (2 dans notre exemple) sera lue et que la validité de cet argument sera contrôlée puis que l'instruction sera exécutée. On retourne ensuite dans l'interpréteur: le processus décrit se répète à nouveau. La tâche que remplit dans notre exemple Mme Interpréteur nécessite bien sûr un certain temps. Ce temps est économisé lorsqu'on programme directement en langage-machine.

Malheureusement, le langage-machine présente l'inconvénient d'être très abstrait. L'homme a en principe du mal à se représenter des nombres. C'est pourquoi des langages de programmation dit évolués, tels que Logo ou Basic ont été développés. Ces langages évolués travaillent en effet avec des concepts au lieu de travailler avec

des chiffres. Ces langages représentent un compromis dans la communication entre l'homme et la machine. Malheureusement ces langages ont des inconvénients en ce qui concerne la vitesse d'exécution, la place occupée en mémoire et souvent également en ce qui concerne les possibilités de programmation.

Tous les langages évolués comme également Cobol ou Pascal doivent étre interprétés avant que l'ordinateur ne puisse les exécuter. On distingue à cet égard entre Interpréteur et Compilateur :

Un interpréteur, comme par exemple celui du CPC traduit toutes les instructions les unes après les autres lors de chaque exécution du programme et il les exécute immédiatement. L'interpréteur est donc un traducteur en simultané, c'est-à-dire que chaque instruction est à nouveau traduite chaque fois que le programme est exécuté. C'est pourquoi la modification d'un programme Basic pose aussi peu de problèmes.

Au contraire, un compilateur n'interpréte le programme qu'une seule fois en produisant en même temps un équivalent en langage-machine. C'est alors seulement que peut être exécuté le code machine ainsi produit. Si le programme est modifié, la nouvelle version doit à nouveau être compilée. C'est pourquoi la modification de tels programmes est assez longue. Nous vous présenterons dans ce livre un compilateur qui effectue une traduction du langage-assembleur en code machine. On appelle un tel compilateur un ASSEMBLEUR.

Voici donc bien un premier avantage essentiel du langage-machine : les programmes machine peuvent être exécutés jusqu'à 1000 fois plus vite que des programmes Basic. L'instruction RETURN en Basic est exécutée en environ 0.6 millisecondes alors que l'instruction correspondante en langage-machine ne dure que 2.5 microsecondes. Le langage-machine est donc 240 fois plus rapide pour l'instruction RET et l'équivalent en langage-machine de l'instruction POKE est méme 1000 fois plus rapide que celle-ci. Ces différences sont importantes par exemple lorsqu'il s'agit de trier ou d'effectuer des recherches avec des grandes masses de données, ou pour décaler le contenu de sections de la mémoire comme cela est nécessaire pour le scrolling ou pour les programmes de traitement de textes. La programmation du graphisme haute résolution est d'autre part trop

lente en Basic, de sorte que le langage-machine s'impose pour les jeux ou les graphiques de gestion par exemple.

Le langage-machine possède encore de nombreux avantages.

En règle générale, les programmes machine sont plus courts que les programmes Basic ce qui permet d'économiser beaucoup de place en mémoire. Dès que vous aurez écrit vos premiers programmes en langage-machine, vous constaterez qu'un programme machine de plus de 500 octets est déjà très long et qu'il permet déjà de faire un grand nombre de choses. Par contre il faudrait beaucoup plus de place en mémoire pour stocker un programme Basic de fonction équivalente.

Remarque : la longueur d'un programme Basic en octets peut être calculée sur le CPC avec la formule >PRINT HIMEM-FRE(0)-370<.

Un autre avantage essentiel du langage-machine est qu'il permet seul d'utiliser pleinement toutes les possibilités d'un ordinateur. En langage-machine, on est par exemple à même de programmer les composants d'entrée/sortie. On peut donc communiquer avec les périphériques en utilisant des programmes qu'on a développé soi-même.

Le développement de vos propres structures de données, qui occuperont souvent beaucoup moins de place en mémoire que celles offertes par le Basic, n'est également possible qu'en langage-machine. Il est ainsi plus aisé de loger dans la place disponible en mémoire les grandes masses de données qu'on rencontre par exemple en traitement de texte.

Ces exemples devraient suffir à marquer tout l'intérêt de la programmation en langage-machine, même sur un ordinateur disposant d'un Basic aussi puissant que celui du CPC. Il faut cependant également reconnaître que la programmation en langage-machine présente également un grand inconvénient.

Le langage-machine étant le langage le plus proche de la machine, cela présente pour le programmeur l'inconvénient de devoir penser de façon très abstraite pour comprendre ce langage. L'homme pense

avant tout avec des mots et des associations, ce qui est très loin de la structure du langage-machine. En effet un programme machine est simplement constitué d'une suite de nombres et non d'une suite de concepts. Il serait pratiquement impossible de développer des programmes d'envergure sous cette forme. C'est pourquoi les pionniers de l'informatique ont développé une espède de langage intermédiaire qui rend les programmes machine plus clairs et plus compréhensibles, l'ASSEMBLEUR. Le langage assembleur affecte une série de symboles à chaque code machine (soit un nombre). Ces symboles se composent de :

1. un mot d'instruction, la plupart du temps une abréviation du nom anglais de l'instruction, aussi appelée mnémonique.

2. un opérande qui permet par exemple d'indiquer des adresses ou des constantes (se rapportant au mot d'instruction).

La création d'un programme machine peut ainsi se faire en écrivant en langage assembleur. Le langage assembleur est ensuite automatiquement traduit en code machine par ce qu'on appelle un programme d'assembleur. Nous vous présenterons dans ce livre un assembleur que nous utiliserons pour programmer en assembleur. C'est pourquoi nous n'évoquerons que brièvement et uniquement à titre d'exemple la programmation en véritable langage-machine, sous forme de nombres, et que nous passerons vite à la programmation en assembleur, laissant à l'assembleur (le compilateur) le soin d'effectuer le travail d'interprétation.

Et maintenant les choses sérieuses commencent ! ! !

1.2. LE PREMIER PROGRAMME MACHINE

Pour vous montrer l'intérêt qu'il y a à apprendre le langage-machine, voici une comparaison entre un programme Basic et votre premier programme machine :

Veuillez entrer les lignes Basic suivantes :

```
10 HL=&C000
20 POKE HL,&CC
30 HL=HL+1
40 IF HL<=&FFFF THEN 20
50 RETURN
```

Entrez maintenant en mode direct >MODE 2< puis >GOSUB 10< et voyez ce qui arrive !

Le programme suivant charge le programme machine qui exécute la même fonction que le programme Basic :

```
10 MEMORY &9FFF
20 FOR I=&A000 TO &A009
30 READ a
40 POKE i,a
50 NEXT I
60 END
70 DATA &21,&00,&C0,&36,&CC,&23,&BC,&20,&FA,&C9
```

Entrez maintenant à nouveau >MODE 2< en mode direct, chargez le programme avec >RUN<, lancez le programme machine ainsi chargé avec >CALL &A000<. Vous pouvez admirer !

Comme vous avez vu,

- le programme Basic dure environ 1 minute
- le programme machine environ 1/10 de seconde

La durée d'exécution peut être calculée théoriquement. Elle est de 0.1106 secondes pour le programme d'exemple.

La longueur des programmes est de :

- programme Basic : 88 octets
- programme machine : 10 octets
(de &A000 à &A009)

Ne soyez pas dérouté par les choses nouvelles que vous rencontrez ici. Dans les chapitres suivants, tout sera expliqué en détail.

Comparaison des deux programmes :

Basic			Langage assembleur
10	HL=&C000	-	LD HL,C000
20	POKE HL,&CC	-	LD (HL),&CC
30	HL=HL+1	-	INC HL
		-	CP H
40	IF HL<=&FFFF THEN 20	-	JR NZ,$-6>A006
50	RETURN	-	RET

EXPLICATION :

Ligne 10 : la valeur de la VARIABLE HL ou du REGISTRE HL est fixée sur le début de la mémoire écran

Ligne 20 : cette ligne stocke à l'adresse HL la valeur &CC. Comme la mémoire écran est située de &C000 à &FFFF, cette instruction provoque une modification de l'écran.

Essayez maintenant de modifier simplement en mode direct les valeurs pour l'adresse HL en mémoire écran (HL doit être entre &C000 et &FFFF!) et pour l'argument (&CC dans notre programme. Vous pouvez utiliser des valeurs comprises entre &00 et &FF (par exemple : POKE &C100,&AA).

Ligne 30 : Augmente de 1 la variable ou le registre HL.

Ligne 40 : Teste si HL est supérieur à &FFFF, c'est-à-dire si la fin de la zone écran a été atteinte. Ce test doit être décomposé en langage-machine en deux instructions : CP (comparer) et JR (saut relatif) NZ (non zéro), ce qui veut dire :
"Saute, si pas zéro".

Voici maintenant à titre d'exemple le listing assembleur qu'on obtiendra lors de l'assemblage (compilation en code machine) :

LISTING ASSEMBLEUR du programme machine

Adresse Code Ligne Basic Instruction Ass. Commentaire

```
A000 210020     10          LD    HL,C000 ;début mémoire écran
A003 36CC       20          LD    (HL),&CC ;valeur à écrire dans la
mémoire écran
A005 23         30          INC   HL ;HL=HL+1
A006 BC         40          CP    H ;comparer à zéro
A007 20FA       50          JR    NZ,$-6 > A006 ; si pas zéro
retourner de 6 adresses en arrière, si zéro, prochaine instruction
A009 C9         60          RET ;retour au Basic
```

Nous espérons que nous avons pu éveiller votre curiosité et nous allons maintenant passer à l'étude systématique du langage-machine.

1.3. SYSTEMES NUMERIQUES

Dans le chapitre précédent, le signe & a été utilisé comme la marque d'un nombre en hexadécimal (hexadécimal - 16). Qu'est-ce que cela signifie ?

Lors de la réalisation d'installations de calcul électroniques on pouvait représenter les nombres de deux façons.

Analogique : Avec une calculatrice analogique un nombre est représenté par une tension correspondante, par exemple 1 = 1 volt et 100 = 100 volts. Une montre à aiguilles est par conséquent une montre analogique. La progression constante du temps correspond (est analogue à) au nombre de révolutions des aiguilles.

Digitale : Avec les ordinateurs digitaux on ne prend pas en compte la mesure de la tension mais seulement les deux états possibles : le courant circule, le courant ne circule pas. Digital signifie représentation de grandeurs à l'aide de chiffres. Les états ETEINT et ALLUME correspondent alors aux chiffres 0 et 1.

Ainsi un ordinateur digital ne dispose que de deux chiffres. C'est à l'aide de ceux-ci que s'effectue la représentation des nombres dans l'ordinateur.

Pour des tâches bien précises, le traitement avec une calculatrice analogique est, dans certaines circonstances, tout à fait approprié (par exemple commande de machine). Si les problémes les plus divers doivent cependant être résolus sur un ordinateur, l'ordinateur digital est nettement supérieur à la calculatrice analogique car la programmation d'une calculatrice dans la forme que nous connaissons n'est pas possible. Cela signifie que des ordinateurs personnels et familiaux sont des ordinateurs digitaux et traitent ainsi les données en système binaire (avec les chiffres 0 et 1).

Pour le programmeur, les systèmes numériques suivants sont importants :

1. Système décimal
2. Système binaire
3. Système hexadécimal

Les systèmes numériques sont des schémas d'ordre des chiffres construits d'après un principe bien défini. Chaque nombre peut être, après un nouveau calcul, transformé dans un autre système numérique. Dans tous les systèmes numériques la valeur de position d'un chiffre augmente de la droite vers la gauche.

Pour expliquer les autres systèmes numériques, partons du système décimal que l'on utilise couramment.

Le système décimal

Milliers	Centaines	Dizaines	Unités	
Milliers	Centaines	Dizaines	Unités	- Valeur de position
7	3	5	6	- Chiffres

<——————————————————————

La valeur de position augmente de la droite vers la gauche !

Puissance	Chiffre	Nom
10^0	1	U-nité
10^1	10	D-izaine
10^2	100	C-entaine
10^3	1000	M-illier
10^4	10000	Dizaine de milliers
10^6	1000000	Million

On peut aussi écrire le nombre 1335 de la façon suivante :

1335 signifie : 1M + 3C + 3D + 5U - La plus petite valeur de position (Unité) se trouve

435 signifie : 4C + 3D + 5U - à l'extrême droite.

1335 est : 1*1000+3*100+3*10+5*1

1335 est aussi : $1*10^3 + 3*10^2 + 3*10^1 + 5*10^0$

On définit une puissance avec l'exposant 0 comme valant 1.

0 0 0

par ex. : 10 = 1, 2 = 1, x = 1

Le système binaire

Le système binaire est construit sur le même principe. La différence réside dans le fait que la valeur de position des différents chiffres n'est pas représentée par des puissances de dix mais par des puissances de deux.

La base du système binaire est 2.

Binaire 10101101 = Décimal 173

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	- Valeur de position
1	0	1	0	1	1	0	1	- Chiffre

$173 = 1*2^7 + 0*2^6 + 1*2^5 + 0*2^4 + 1*2^3 + 1*2^2 + 0*2^1 + 1*2^0$

$173 = 1*128 + 0*64 + 1*32 + 0*16 + 1*8 + 1*4 + 0*2 + 1*1$

Jusqu'à maintenant vous avez appris la conversion à partir du système binaire dans le système décimal. On peut bien sûr inverser aussi ce processus. Pour l'expliquer, considérons le nombre décimal 173 calculé plus haut.

Nous nous demandons quelle puissance de 2 est justement contenue dans ce nombre. Nous vous aidons : En principe on peut appliquer le système binaire à des nombres à n chiffres. Mais en informatique, on utilise seulement des nombres binaires à 8 chiffres. Les puissances de 2 suivantes peuvent se présenter:

Puissances de 2	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Valeurs converties	128	64	32	16	8	4	2	1

Dans ce cas c'est 2^7=128 qui est la puissance de 2 la plus haute. Calculons maintenant la différence entre 173 et 128. Le résultat donne 45. On procède alors avec ce reste de la même façon que précédemment. Nous cherchons alors à nouveau la puissance de 2 la plus haute qui se trouve dans cette valeur. A l'aide du tableau on peut la déterminer facilement, elle est de 2^5=32.

Puis nous calculons à nouveau la différence : (45-32=13).

Le procédé décrit sera appliqué jusqu'à ce que le reste soit égal à zéro.

```
2^3 = 8      (13 - 8 = 5)
2^2 = 4      ( 5 - 4 = 1)
2^0 = 1      ( 1 - 1 = 0)
```

Nous avons obtenu les puissances suivantes de 2 :

```
2^7 , 2^5 , 2^3 , 2^2
```

Sous chaque puissance de 2 rencontrée, nous écrivons un Un et sous celles qui manquent un zéro :

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0		
1	0	1	0	1	1	0	1	=	173

Le nombre décimal 173 est donc représenté en système binaire par 10101101. Par la suite nous allons désigner les nombres binaires en les faisant précéder par &X.

Par ex. : 173= &X 10101101

Bit et octet

Un BIT est la plus petite unité d'information dont se composent toutes les autres informations. BIT est l'abréviation de "binary digit", ce qui veut dire à peu de choses près chiffre binaire. On dit qu'un BIT est mis lorsque le BIT a l'état 1 ou qu'un BIT est annulé lorsqu'il a l'état 0.

Le CPC 464 a un processeur 8-BITS, c'est-à-dire qu'il peut traiter des nombres binaires longs de 8 BITS, ce qui correspond aux valeurs décimales de 0 à 255.

Nombre binaire :

1 0 1 1 0 1 1 1

m a m m a m m m   m = BIT mis ; a = BIT annulé

7 6 5 4 3 2 1 0   Numéro du BIT

Chaque bit (chaque chiffre) d'un nombre binaire reçoit un numéro de bit. Le bit à la valeur de position la plus faible, c'est-à-dire celui se trouvant à l'extrême droite, porte le numéro 0. On continue ainsi la numérotation de droite à gauche. Le numéro de bit correspond à l'exposant de la puissance 2 qui représente la valeur de position correspondante.

En informatique il convient de s'imaginer les états de BIT comme un commutateur.

COMMUTATEUR ALLUME = 1

COMMUTATEUR ETEINT = 0

Dans le cas d'un nombre de 8 commutateurs on peut imaginer des valeurs de 0 à 255, donc 256 états de commutateurs.

On appelle huit commutateurs (BITs) réunis un OCTET. Un octet peut être placé dans une mémoire par l'ordinateur. Mais que deviennent les nombres supérieurs à 255 à A cet effet on divise le nombre en deux moitiés, à savoir un octet faible et un octet fort. Ces deux octet sont ensuite placés dans deux cellules successives de la mémoire.

On peut calculer l'octet faible et l'octet fort de la manière suivante :

Nombre divisé par 256 = (octet-fort) + reste
Le reste de la division correspond à l'octet faible.

Pour mémoire : le nombre 255 est la valeur maximum pouvant être représentée dans un octet puisqu'il se compose de 8 BITs.

Exemple : Le nombre 34065 doit être divisé en un octet faible et un octet fort.

```
34065 / 256    = 133 reste 17
34065          = 133 * 256 + 17
```

```
133 = octet fort
 17 = octet faible
```

La formule générale écrite en BASIC donne :

```
OFO = INT(Nombre/256)        OFO = Octet FOrt
OFA = Nombre-OFO*256         OFA = Octet FAible
```

Ainsi un nombre entre 256 et 65535 placé en mémoire nécessite 2 octets.

Pour représenter plus facilement les nombres qui sont mis en mémoire, il est intéressant d'introduire un autre système numérique.

Le système hexadécimal

Dans le système hexadécimal la base est 16.
Pour mémoire :

Dans le sytème décimal la base est 10.
Dans le système binaire la base est 2.

Pour représenter les chiffres dont la valeur est supérieure à 10, on utilise dans le système hexadécimal les lettres A à F.

```
Système décimal :
0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,...

Système hexadécimal :
0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,...
```

Nous convertissons d'abord les nombres hexadécimaux en nombres décimaux :

Puissance	Valeur
16^0	1
16^1	16
16^2	256
16^3	4096

```
&3ABF = 3*16^3   + 10*16^2   + 11*16^1   + 15*16^0
&3ABF = 3*4096   + 10*256    + 11*16     + 15* 1
&3ABF = 12288    + 2560      + 176       + 15
&3ABF = 15039
```

Autre exemple :

```
&1A3E = 1*16^3   + 10*16^2   + 3*16^1    + 14*16^0
&1A3E = 1*4096   + 10*256    + 3*16      + 14*1
&1A3E = 4096     + 2560      + 48        + 14
&1A3E = 6718
```

L'avantage du système hexadécimal réside dans le fait que l'on peut lire directement l'octet fort et l'octet faible.

Pour &3ABF :

- l'octet fort se compose des deux premiers chiffres hexadécimaux (3 et A). Il a la valeur décimale (3*16^1+10*16^0) = 58.

- l'octet faible se compose des deux derniers chiffres hexadécimaux (B et F). Il a la valeur décimale (11*16^1 + 15*16^0) = 191.

Effectuez l'entrée suivante :

```
PRINT PEEK(9),PEEK(10)
```

Aux deux adresses 9 et 10 on trouve l'adresse de saut où le système d'exploitation saute lorsqu'une routine doit être appelée dans la ROM inférieure.

Pour une adresse de saut on peut envisager une valeur de 0 à 65535 (donc jusqu'à &FFFF). Ce nombre est sauvegardé à l'aide des octets fort et faible. Nous allons maintenant calculer l'adresse de saut. Avec l'instruction BASIC ci-dessus nous obtenons à l'adresse 9 la valeur 130 et à l'adresse 10 la valeur 185.

Attention : Ces valeurs peuvent être sensiblement différentes selon le modèle de votre ordinateur CPC 464,664,6128.

Dans le mode décimal l'adresse de saut donne donc 185*256+130 = 47490.

Nous allons maintenant exécuter le même calcul dans le système hexadécimal :

130 = &82 et 185 = &B9, ainsi que vous pouvez le vérifier aisément. Nous obtenons facilement la valeur de l'adresse de saut en écrivant l'un derrière l'autre l'octet fort et l'octet faible : 47490 = &B982.

Il est tout aussi facile de diviser un nombre hexadécimal en octet fort et faible que de le composer de cette façon. En général, l'octet faible d'un nombre se trouve à l'adresse inférieure de la mémoire, vient ensuite l'octet fort.

Vous avez pu vous rendre compte ainsi du premier avantage du système hexadécimal. En outre la conversion du système binaire en systéme hexadécimal est facile à exécuter. A cet effet on divise un nombre binaire en deux blocs de 4 bits chacun. Le bloc des bits 0 à 3 est appelé quartet inférieur et l'autre bloc, des bits 4 à 7, quartet supérieur. Chaque quartet correspond exactement à un chiffre hexadécimal. C'est facilement reconnaissable car un nombre binaire à 4 bits peut prendre la valeur maximum de 15 (15 = 8 + 4 + 2 + 1) et toutes les valeurs de 0 à 15 peuvent aussi être représentées par un chiffre hexadécimal (0,1,...,9,A,B,C,D,E,F).

Considérons un exemple :

1 1 0 1	1 0 0 1
Q. fort	Q. faible
8+ 4+ +1	8+ 1
13	9
&D	&9

Alors : &X11011001=&D9

En vous exerçant un peu vous pourrez reconnaître directement dans un nombre à 4 bits le chiffre hexadécimal correspondant et inversement. Le tableau suivant est censé vous y aider :

Système binaire	hexadécimal	décimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

La conversion de l'hexadécimal vers le binaire s'effectue de façon similaire. Chaque chiffre hexadécimal est remplacé par la combinaison correspondante à quatre bits, par exemple &C7 = &X1100 0111.

Comprendre la conversion entre les différents systèmes numériques est une nécessité de la programmation en langage machine.

Exercices :

1. Complétez les tableaux suivants :

Décimal	Binaire	Hexadécimal
130	?	?
?	10010011	?
57312	----	?
?	----	&COB6
?	?	&37

2. A partir de l'adresse &A000 de la mémoire la valeur 37315 doit être mémorisée. Calculez l'octet fort et l'octet faible et indiquez les instructions BASIC à partir desquelles le nombre peut être mémorisé.

3. A partir de l'adresse &0006 se trouve une adresse de saut importante du système d'exploitation. Quelle est sa valeur ?

Solutions :

1.

Décimal	Binaire	Hexadécimal
130	10000010	&82
147	10010011	&93
57312	----	&DFE0
49334	----	&COB6
55	00110111	&37

2. Octet fort = 145 = &91 ; octet faible = 195 = &c3
POKE &A000,&C3:POKE &A001,&91

3. Octet faible = PEEK(&0006), octet fort = PEEK(&0007)
Adresse de saut = &0580

Attention : Ici aussi, cette valeur peut varier d'un modèle d'ordinateur à l'autre.

En annexe vous trouverez un tableau dans lequel sont indiqués les nombres allant de 0 à 255 (1 octet) dans les trois systèmes numériques.

1.4. STRUCTURE DE L'ORDINATEUR

Si nous voulons traiter de la programmation en langage-machine, il faut que nous ayons une idée de l'organisation et de la structure internes de l'ordinateur. Nous allons donc vous livrer une présentation de l'ordinateur qui suffise à nos besoins.

Comme vous le savez, vous possédez un ordinateur 64K ou 128 K si vous possédez le 6128 (K = kilo - octets = 1024 octets). Cela signifie que la capacité de mémoire de l'ordinateur est de 64*1024 = 65536 octets ou 131072 octets pour un 6128. Comme un octet se compose de 8 bits et que c'est ainsi que l'ordinateur représente les données, votre ordinateur se compose donc de 64*1024*8 bits, soit environ de 500 000 commutateurs (le double pour un 6128) qui sont soit ouverts soit éteints. Par la suite, nous ne parlerons plus que de 64 K de mémoire car même si vous possédez un 6128, les 64 K supplémentaires sont très difficiles à utiliser. Il est cependant difficile de se représenter ainsi le travail concret avec l'ordinateur. C'est pourquoi les groupes de 8 bits sont réunis pour former un octet. Ces 64*1024 octets figurent dans la Ram de votre ordinateur. Ram signifie Random Access Memory, ce qui signifie qu'il s'agit d'une mémoire d'écriture et de lecture ou d'une mémoire de travail. Les 65536 octets de la Ram sont numérotés de &0000 à $FFFF. Le numéro correspondant à un octet est son adresse. Cette adresse est normalement indiquée sous forme d'un nombre hexadécimal. Il est possible en Basic d'accéder directement à la Ram. C'est à cela que servent les instructions PEEK et POKE. >PEEK(adresse)< lit la valeur de l'octet figurant à l'adresse indiquée, et >POKE adresse,valeur< stocke la valeur indiquée dans l'adresse indiquée.

Comme chaque adresse est affectée à un octet et qu'un octet se compose de 8 bits, et a donc une valeur comprise entre 0 et 255 (&00-&FF), la valeur à stocker doit être également comprise entre ces limites. L'adresse doit bien sûr être également comprise entre &0000 et &FFFF.

La Ram sert au stockage des programmes que vous entrez. En outre,le contenu codé de l'écran est stocké à partir de &C000. En mode 2 un point est représenté par un bit mis. Enfin quelques routines importantes du système d'exploitation ainsi que des informations sur les couleurs actuelles, l'affectation des touches, les caractères définis par l'utilisateur, etc. se trouvent en Ram. Comme des routines système ainsi que des informations importantes se trouvent dans la Ram, des POKEs inconsidérés peuvent conduire à "planter" l'ordinateur. n'essayez par exemple jamais >POKE &8,0<.

La Ram est ainsi organisée :

&0000-&0170	utilisé par le système
&0171-&AB7F	pour les programmes Basic
&AB80-&BFFF	utilisé par le système
&C000-&FFFF	mémoire écran

En réalité, les adresses 80170 et &AB7F peuvent varier selon la configuration et le modèle de votre ordinateur. La valeur 80170 n'a en fait, pas grand intérêt et la valeur exacte de l'adresse de début de la RAM utilisée par le système est donnée par PRINT HIMEM + 1.

Il est possible de limiter la place réservée pour les programmes Basic, avec l'instruction >MEMORY adresse<. Nous pouvons ainsi disposer pour le stockage de nos programmes en langage-machine, de la zone comprise entre l'adresse indiquée dans l'instruction MEMORY et &AB7F. Nous pouvons par exemple réserver pour notre programme en langage-machine la zone de &A000 à &AB7F, avec l'instruction >MEMORY &9FFF<. Nous pouvons ensuite stocker notre programme en langage-machine à partir de &A000 avec des instructions POKE.

Vous vous étonnez peut-être que seul un peu plus de 1K soit utilisé pour des routines système.

Où se trouvent donc l'interpréteur et le système d'exploitation qui nous permettent de programmer en Basic ?

Vous avez raison :

Il y a encore une mémoire importante, la Rom (Read Only Memory = mémoire de lecture uniquement). C'est dans la Rom que se trouvent toutes les données et les programmes qui nous permettent de programmer si aisément en Basic. Comme une Rom est une mémoire fixe, elle reçoit des données et des programmes (en langage-machine !) et elle est intégrée dans l'ordinateur à l'usine. Malheureusement, il ne nous est pas possible de lire à partir du Basic, le contenu de la Rom.

Le CPC possède deux Roms de 16K dont les adresses chevauchent celles de la Ram. Cela est rendu nécessaire, du fait que le processeur Z80 ne possède que 16 bits d'adresse et que donc l'adresse d'un octet ne peut comporter plus de 16 bits. 16 bits couvrent exactement la zone entre &0000 et &FFFF. Pour lire la Rom, il faut donc indiquer à l'unité centrale qu'il s'agit de lire la Rom, après quoi il est possible d'utiliser les mêmes adresses que pour la Ram. Les Roms occupent les zones suivantes:

1.	Rom &0000 - &3FFF	Système d'exploitation
2.	Rom &C000 - &FFFF	Basic

Le système d'exploitation contient les routines fondamentales nécessaires pour que l'ordinateur puisse travailler. Il est donc responsable de la commande des périphériques, de la gestion des données, du transfert de données, etc... Dans la zone de Rom inférieure figurent également les copies des routines système qui se trouvent en Ram. Lors de la mise sous tension ou lors d'un Reset de l'ordinateur, ces routines sont copiées de la Rom dans la Ram. La mémoire de caractères, où chaque caractère de l'ordinateur est représenté dans une matrice de bits (0 = pas de point, 1 = point), se trouve également dans la Rom (&3800-&3FFF).

Les instructions Basic que nous programmons sont exécutées par les programmes figurant dans la Rom Basic. La table des mots-instructions figure par exemple en &E388.

Notre ordinateur comprend bien sûr de nombreux autres circuits intégrés, comme le processeur Z80 ou le chip sonore. Nous décrirons le processeur Z80 dans le chapitre suivant. Si vous êtes intéressé par de plus amples informations sur la structure interne de votre ordinateur, reportez-vous à l'ouvrage "La bible du CPC".

Sur le 6128, il y a encore une autre zone de mémoire ; ce sont les 64 K supplémentaires. Cependant, cette mémoire est très difficile à gérer. Un utilitaire nommé BANKMAN et se trouvant sur la disquette C/M livrée avec votre CPC 6128, permet la gestion de ces 64 K supplémentaires. Des exemples d'utilisation de ces programmes sont donnés dans "Le grand livre du Basic 6128" de la collection MICRO APPLICATION.

2. Le processeur Z80

2.1. STRUCTURE DE L'UNITE CENTRALE

(voyez l'illustration 1:Chapitre 2.1)

Le CPC possède une unité centrale Z80. Nous nous souvenons que *l'unité centrale peut être considérée comme le* cerveau de l'ordinateur. La signification de cette MPU (MPU: angl: Micro Processing Unit - microprocesseur) est donc claire.

Dans ce chapitre, nous traiterons de la structure et de la fonction des divers composants que contient l'unité centrale. L'illustration correspondant à ce chapitre doit nous aider à comprendre le fonctionnement interne d'une unité centrale. Si nous examinons l'illustration de gauche à droite, nous reconnaissons :

1. **Cu (Control Unit= unité de contrôle)**

   Toutes les opérations qui se déroulent dans un ordinateur sont contrôlées et commandées par la CU.

2. **Bus de contrôle**

   Le bus de contrôle est le "bras long" de la CU. C'est lui qui pilote et surveille les composants extérieurs à l'unité centrale.

3. **Le pointeur de pile SP (Stack Pointer)**

   Le pointeur de pile permet le stockage provisoire dans la Ram des données et des adresses de retour des sous-programmes. Comme il est possible de stocker des adresses dans le SP, il s'agit d'un registre 16 bits.

4. **Pointeur de programme PC (Programm Counter = compteur de programme)**

Le PC indique l'adresse de la mémoire où figure l'instruction à traiter à un moment donné.

5. **Registres B à L**

L'unité centrale possède plusieurs registres dans lesquels les données peuvent être stockées.

6. **Flags (flag = drapeau ; ici cela signifie plutôt marque)**

Les flags servent d'indicateurs pour certains événements qui se produisent lors des opérations de calcul à l'intérieur de l'unité centrale. Les flags peuvent être mis (drapeau levé) ou annulés (drapeau baissé).

7. **Le bus d'adresse (se trouve en dehors de l'unité centrale)**

Le bus d'adresse crée le lien avec les autres MPUs de l'ordinateur. Il indique la case mémoire de la Rom ou de la Ram dont le contenu doit être lu ou modifié. Le bus d'adresse a une taille de 16 bits, ce qui est nécessaire pour pouvoir adresser une mémoire d'une taille de 64K.

8. **Le bus de données (se trouve en dehors de l'unité centrale)**

Les bus de données "fournissent" les données lues ou à écrire. Le bus d'adresse indique pour cela l'adresse des données. Le bus de données a une largeur de 8 bits.

9. **Accumulateur**

L'accumulateur (accu) est le registre le plus important de l'unité centrale. On peut également le qualifier de registre de calcul.

10. ALU (Arithmetical Logical Unit = unité arithmétique et logique ou unité de calcul)

L'ALU exécute toutes les opérations arithmétiques et logiques. Les flags sont modifiés en fonction du résultat des opérations.

11. Unité de décalage

L'unité de décalage exécute les opérations de rotation et de décalage.

Comme nous l'avons déjà indiqué pour le point 5, l'unité centrale contient plusieurs registres. Pour éclairer leurs différentes fonctions, nous les avons répartis en 5 groupes :

1. L'accumulateur
2. Les flags
3. Les 6 registres 8 bits qui peuvent être couplés.
4. Les quatre registres 16 bits indivisibles
5. Registre d'interruption et de refresh

2.2. L'ACCUMULATEUR

L'accumulateur ou registre A est le registre le plus important du Z80. La plupart des instructions arithmétiques et logiques utilisent ce registre. Pour exécuter une instruction de comparaison, c'est avec le contenu de l'accumulateur que s'effectue la comparaison. Comme tous les registres exceptés SP, PC, IX et IY, le registre A est un registre 8 bits.

2.3. LES FLAGS

Le registre F ou registre Flags a une taille de 8 bits (comme A,B,C,D,E,H et L). Il a cependant des fonctions différentes de celles de ces registres.

Dans le registre flags, les différents bits sont utilisés comme indicateurs de certains événements qui se produisent lors d'opérations de l'ALU (l'unité de calcul). Les différents bits du registre F ont la signification suivante :

S	Z		H		P/V	N	C	- désignation du flag
7	6	5	4	3	2	1	0	- numéro du bit

C - Carry = retenue
N - soustraction
P/V - parité/dépassement
H - demi-retenue
Z - Zéro
S - Signe

Flag C (bit 0)

Si une retenue se produit lors d'une addition ou d'une soustraction, ce bit est mis, sinon il est annulé.

Flags N et H (bit 1, bit 4)

Ces flags sont utilisés par le Z80 de façon interne. Il n'ont pas de signification pour nous pour le moment.

Flag P/V (bit 2)

Ce flag a une double fonction :

Il est mis lorsqu'un dépassement (V) (angl.: oVerflow) se produit, sinon il est annulé. Il indique par ailleurs la parité d'un octet (P).

Flag Z (bit 6)

Ce flag est mis lorsque le résultat d'une soustraction est zéro, sinon il est annulé. Pour une comparaison, ce bit est mis lorsqu'il y a identité.

Flag S (bit 7)

Si le résultat d'une addition ou d'une soustraction est supérieur à 127, ce bit est mis. Comme nous le verrons plus tard, pour l'arithmétique de l'unité centrale, les octets supérieurs à 127 représentent des nombres négatifs.

Les bits 3 et 5 du registre flags ne sont pas utilisés.

2.4. LES 6 REGISTRES 8 BITS QUI PEUVENT ETRE COUPLES

Ces six registres 8 bits sont : B, C, D, E, H, L

Ces registres peuvent être regroupés par paires pour former des registres 16 bits. C'est respectivement dans C, E et L que sont stockés les octets faibles et dans B, D et H les octets forts.

B/C (Byte Counter)

Le registre B ou la paire de registres BC sont souvent utilisés comme compteur, par exemple pour les boucles.

La paire de registres DE peut être librement utilisée.
Cette paire de registres est souvent employée pour le stockage intermédiaire d'adresses ou de données.

H/L (High/Low)

La paire de registre HL est souvent employée pour y stocker des adresses.

Il est recommandé de se familiariser avec les noms de ces registres car certaines instructions utilisent les registres avec les noms que nous vous indiquons ci-dessus. En principe, on peut également utiliser le registre L ou le registre E comme compteur.

Une particularité du Z80 réside dans le fait que les registres indiqués ci-dessus existent en double avec les mêmes fonctions. Ce deuxième jeu de registres est à notre disposition, mais seul un jeu de registre peut être utilisé à la fois.

2.5. LES 4 REGISTRES 16 BITS INDIVISIBLES

A ce groupe appartiennent 4 registres 16 bits :

SP, PC, IX, IY

Le registre SP est un registre de 16 bits fixe qui ne peut donc pas être séparé en deux registres 8 bits. Le pointeur de pile SP indique en permanence dans quelle adresse de la mémoire figurent les adresses de retour ou les données stockées provisoirement. Cette adresse désigne une case mémoire qui se trouve dans une zone de la Ram appelée pile. Le stockage de données sur la pile s'effectue ainsi :

Lors de la mise sous tension de l'ordinateur, le pointeur de pile SP est fixé sur la plus haute adresse de la pile (&C000). Si un octet doit ensuite être placé sur la pile, SP est automatiquement diminué de 1 et cet octet est stocké dans l'adresse indiquée par le pointeur de pile SP. Le pointeur indique donc toujours la dernière entrée sur la pile. Lorsqu'on retire des données de la pile, le processus est exactement inverse. L'octet figurant à l'adresse indiquée par le pointeur de pile est d'abord lu puis le pointeur de pile est augmenté de 1. Il est ainsi possible d'imbriquer des appels de sous-programmes autant qu'on le souhaite.

Le PC est un registre particulier. Il n'est pas possible de le modifier par programmation.

Les registres IX/IY sont essentiellement utilisés pour le stockage d'adresses ou d'adresses relatives. Il n'est pas possible d'accéder séparément aux octets faible ou fort de ces registres 16 bits, contrairement à ce qui est le cas pour les registres BC, DE, HL.

L'utilisation des registres d'index est semblable à celle de la paire de registres hl. Nous expliquerons la différence entre ces deux types de registres lorsque nous étudierons l'adressage indexé.

2.6. REGISTRE D'INTERRUPTION ET DE REFRESH

Ces deux registres sont affectés à l'unité centrale.

Registres I ou registre d'interruption.

Si une interruption se produit, ce registre 8 bits contient la partie supérieure de l'adresse à laquelle le programme doit sauter. La partie inférieure est fournie par le composant de l'ordinateur qui a provoqué l'interruption.

Registre R ou registre de refresh. (refresh= rafraîchir)

Ce registre est utilisé par l'électronique comme compteur pour rafraîchir à intervalles réguliers le contenu des mémoires dynamiques. Cela permet d'éviter que des informations stockées ne soient perdues. Par le chargement renouvelé en très peu de temps du même contenu de mémoire, on évite une perte de données.

L'exécution d'une instruction par l'unité centrale se présente donc ainsi :

L'octet figurant à l'adresse indiquée par le PC est lu et le PC est augmenté de 1. Il indique donc maintenant l'adresse de l'octet suivant. L'octet lu est interprété comme instruction. On lit alors s'il y a lieu les données qui vont avec l'instruction (le PC est alors à nouveau augmenté. L'instruction est alors exécutée et le processus recommence.

Maintenant que nous connaissons mieux l'unité centrale Z80, intéressons-nous aux instructions en langage-machine.

3. Le jeu d'instructions du Z80

3.1. INTRODUCTION : ENTREE DE PROGRAMMES EN LANGAGE-MACHINE

Pour que nous puissions essayer immédiatement les instructions en langage-machine, nous devons d'abord nous demander comment un programme en langage-machine peut être entré et stocké à partir du Basic. De même qu'en Basic un numéro de ligne est affecté à chaque instruction, une adresse est affectée à chaque instruction en langage-machine.

Basic		Langage-machine		
No ligne	Instruction	Adresse	Instruction	Code
9	HL=HL+1	&A009	INC HL	&23
10	RETURN	&A00A	RET	&C9

- En Basic, un numéro de ligne est affecté à une instruction

- En langage-machine, une adresse est affectée à chaque instruction

Un programme en langage-machine est donc une suite de codes d'instruction qui figurent dans des adresses consécutives de la mémoire.

A partir du Basic, il est possible, avec l'instruction POKE d'écrire les codes dans les adresses voulues. On peut alors appeler un programme en langage-machine avec l'instruction >CALL adresse< où adresse désigne l'emplacement de la mémoire où se trouve la première instruction. Pour que notre programme en langage-machine ne soit pas malencontreusement effacé, il faut que nous lui réservions une zone de la mémoire avec l'instruction MEMORY. Nous utiliserons toujours >MEMORY &9FFF< pour réserver la zone qui va de &A000 à &AB7F. Nous disposerons ainsi de &B80 octets, soit 3K pour nos programmes en langage-machine.

Un programme Basic classique pour charger des programmes en *langage-machine présente la structure suivante :*

```
10 MEMORY &9FFF
20 FOR I=adresse de départ TO adresse de fin
30 READ A
40 POKE I,A
50 NEXT I
60 DATA .....
70 DATA .....
   .
   .
   .
```

En lignes DATA se trouvent les codes qui constitueront le programme en langage-machine proprement dit. L'adresse de fin (V) (nous utiliserons à l'avenir l'abréviation V pour variable que nous placerons à la suite des mots représentant des variables) doit bien sûr être supérieure à &9FFF et l'adresse de départ (V) doit être inférieure à &AB80. Le programme chargé peut être appelé avec l'instruction >CALL adresse de départ<.

Nous utiliserons en principe &A000 comme adresse de départ. L'adresse de fin (V) est donnée par l'adresse de départ (V) plus la *longueur en octets du programme moins* 1. La longueur d'*un programme* correspond au nombre d'entrées en lignes de DATA.

Pour entrer de petit programmes, le programme Basic suivant est pratique :

```
10 CLS
20 MEMORY &9FFF
30 LOCATE 10,10:INPUT"Adresse de depart";adr
40 IF adr <&A000 OR adr>&ABFF THEN 30
50 PRINT
60 PRINT HEX$(adr,4);":";
70 INPUT valeur$
80 IF valeur$="" THEN END
90 valeur=VAL("&"+valeur$)
100 adr=adr+1
```

```
110 IF adr>&AB7F THEN PRINT "Memoire pleine":END
120 GOTO 60
```

Vous entrez les codes décimaux directement, et le programme se chargera de les "poker" dans la mémoire. Pour l'adresse de départ, il est nécessaire d'entrer le signe hexa (&) avec l'adresse. Si vous voulez mettre fin au programme, entrez ENTER.

Maintenant que nous savons comment entrer des programmes en langage-machine, il est temps que nous fassions connaissance avec les instructions du Z80.

Remarque: pour expliquer les instructions, nous aurons souvent recours à des comparaisons avec les instructions Basic. A cet effet, nous nous représenterons les registres en Basic comme étant des variables portant le même nom que le registre en langage-machine (le registre HL en langage-machine correspond à la variable HL en Basic).

Les instructions du Z80 peuvent être classées en 5 groupes:

1. Transfert de données
2. Traitement de données et tests
3. Sauts
4. Instructions de commande
5. Entrée et sortie

3.2. TRANSFERT DE DONNEES

Ces instructions servent au transfert de données :

a) de registre à registre

Cela correspond à l'instruction LET en Basic, comme par exemple A=B ou SP=HL. L'instruction en langage-machine a le format suivant: LD A,B (LD = LOAD, charge)

b) d'un registre à une case mémoire

Pour le transfert de registre à case mémoire, l'instruction Basic >POKE adresse mémoire,variable<, par exemple >POKE &A000,HL< correspond à l'instruction du langage-machine LD(&A000),HL.

c) d'une case mémoire à un registre

Le transfert de données de la mémoire dans un registre, par exemple *LD H,(&A005) correspond à l'instruction* Basic >H=PEEK(&A005)<.

3.3. TRAITEMENT DE DONNEES ET TESTS

Les instructions de traitement de données peuvent être également classées en 5 groupes :

- opérations arithmétiques (par exemple ADDition, soustraction)
- opérations logiques (par exemple AND, OR)
- instructions de comptage (INCrease= augmente, DECrease= diminue)
- manipulation de bits (SET, RESet)
- échange et décalage de bits (Rotate = faire subir une rotation, Shift = décaler)

Lors de l'exécution de ces instructions, le contenu des registres ou de la mémoire (Ram) se modifie. De nombreuses instructions sont semblables à des instructions Basic :

Assembleur	*Basic*
SUB A,B	A=A-B
ADD HL,BC	HL=HL+BC
AND C	A=A AND C
OR (HL)	A=A OR PEEK(HL)

On teste les différents bits des registres ou cases mémoire (instruction BIT) ou on compare le contenu des registres ou de la mémoire avec l'accumulateur (instruction CP=compare). Suivant le résultat de ces tests, les flags correspondants du registre flags sont mis ou annulés par l'ALU.

3.4. SAUTS

Cette instruction permet d'intégrer des sauts dans les programmes en langage-machine.

On distingue trois types fondamentaux :

- saut direct à une adresse 16 bits (JP=jump)
- saut relativement à l'adresse actuelle (JR=jump relative)
- saut à un sous-programme (CALL et retour avec RET)

On qualifie un saut de saut conditionnel, lorsque la décision de sauter ou non dépend de l'état d'un flag. L'instruction JR NZ,$-6>A000 est par exemple un saut conditionnel.

Comparaisons :

Assembleur	Basic
JP	GOTO
CALL	GOSUB
RET	RETURN
JR	---

3.5. INSTRUCTIONS DE COMMANDE

Ces instructions permettent par exemple d'interrompre un programme ou de commander les interruptions.

3.6. INSTRUCTIONS D'ENTREE/SORTIE (Input/Output)

Les instructions I/O sont prévues pour permettre la commande des appareils d'entrée/sortie. Nous présenterons ces instructions par souci d'exhaustivité, mais nous n'en expliquerons pas l'utilisation.

4. Les instructions

4.1. INSTRUCTIONS DE TRANSFERT SUR 8 BITS

Toutes les instructions de transfert de ce type sont représentées par l'instruction de chargement LD.

Une instruction de chargement a le format :

```
LD objet,source
```

Pour les instructions de transfert sur 8 bits, 8 bits sont chargés de la source dans l'objet. Nous allons prendre ces instructions comme exemple pour découvrir les modes d'adressage du Z80.

Chaque instruction en langage-machine se compose d'un code d'opération (opcode) qui peut être suivi d'un opérande ou d'une adresse. Le code d'opération détermine quelle opération doit être exécutée. Un code d'opération contient parfois des bits qui sont utilisés comme pointeur sur un registre. Ces bits ne font donc pas partie à proprement parler du code d'opération. Mais pour simplifier, nous ferons comme si les pointeurs éventuels faisaient partie du code d'opération. Pour certaines instructions, le code d'opération est suivi d'octets de données ou d'adresse. Il y a par ailleurs des instructions dont le code d'opération est long de deux octets. Une instruction peut donc avoir une longueur de 1 à 4 octets.

Pour interpréter les données ou adresses qui suivent une instruction, il est nécessaire de connaître les différents modes d'adressage.

Adressage immédiat

C'est le mode d'adressage le plus simple.

Format :

```
LD reg,data
```

Pour cette instruction, 'reg' représente un registre (A,B,C,D,E,H ou L) et 'data' représente un nombre 8 bits (constante). La constante suivant immédiatement reg est donc chargée dans le registre indiqué reg. Une telle constante est également qualifiée *de littérale.* Le code d'opération sur 8 bits est suivi d'un littéral de 16 bits (la constante).

Exemple :

```
LD C,&7F          Basic: C=&7F
                  (signigie: charge &7F dans registre C)
```

Adressage implicite ou adressage de registre

Les instructions qui travaillent exclusivement avec des registres utilisent l'adressage implicite.

Format

```
LD reg,res
```

Transfére le contenu du registre source res dans reg. Les registres peuvent être A, B, C, D, E, H ou L.

Le nom de ce mode d'adressage vient du fait que l'opérande (c'est-à-dire les deux registres concernés) n'est pas indiqué de façon distincte. C'est en effet le code d'opération qui contient (qui *implique)* les noms des registres concernés.

Le code d'opération de cette instruction en forme binaire est :

```
01OOOSSS
```

Chaque lettre O ou S représente un bit. Les trois O représentent le code du registre objet reg et les trois S représentent le code du registre source res.

Le code des registres est :

```
A-111    E-011
B-000    H-100
C-001    L-101
D-010
```

Exemple :

```
LD B,C =  01 000 001 = &41
          LD  B,  C
```

C'est ainsi que les instructions avec adressage implicite peuvent être représentées par des codes d'opération sur un octet. De ce fait, elles sont exécutées en très peu de temps.

Exemple :

```
LD A,B     Basic: A=B
```

Signifie : transfère le contenu de B dans A.

Zilog Inc. (l'inventeur du Z80) définit le mode d'adressage que nous venons d'étudier comme adressage de registre et réserve le terme d'adressage implicite aux instructions LD I,A ; LD R,A ; LD A,R et LD A,I. Nous ne ferons pas cette distinction et nous utiliserons les termes d'adressage implicite ou d'adressage de registre comme des synonymes.

Adressage absolu ou "étendu"

On qualifie d'adressage absolu le processus qui consiste à aller chercher des données dans la mémoire ou à placer des données dans la mémoire. Avec ce processus, l'adresse 16 bits de la case mémoire (l'adresse absolue) est indiquée entièrement.

Format :

```
LD (adr),reg ou LD reg,(adr)
(adr est l'adresse de la case mémoire)
```

Le contenu de la case mémoire adr est chargé dans le registre reg. La figure 3 vous permet de voir que l'adresse suit le code d'opération. L'adressage absolu nécessite trois octets. Les instructions de ce type sont donc d'une exécution relativement lente.

Exemple :

```
LD A,(&BF93)          Basic: A=PEEK(&BF93)
LD (&A001),A          Basic: POKE &A001,A
```

Adressage indexé

Pour l'adressage indexé, l'adresse de la case mémoire n'est pas fournie de façon absolue mais elle est calculée à partir du contenu d'un registre d'index et d'une distance indiquée.

Format :

```
LD reg,(XY+dis) ou LD (XY+dis),reg
(dis=distance)(XY- un des registres IX ou IY)
```

Chargement de la case mémoire qui a l'adresse suivante dans le registre, ou vice versa: l'adresse est donnée par le contenu du registre d'index et la distance indiquée.

Les instructions indexées possèdent un code d'opération sur deux octets qui est suivi de l'indication de l'adresse. Le premier octet du code d'opération est :

```
&DD - si c'est le registre IX qui est concerné
&FD - si c'est le registre IY qui est concerné
```

Le reste du code est identique, que IX ou IY soit concerné. On utilise la technique de l'adressage indexé pour accéder successivement aux différents éléments d'un bloc de données. La distance peut être positive ou négative, c'est-à-dire que l'octet de distance peut être indiqué comme complément à deux. Mais de toute façon, le registre d'index est toujours augmenté (de façon positive ou négative).

Exemple :

```
LD E,(IX+&32)        Basic: E=PEEK(IX+&32)
LD (IY+&12),A        Basic: POKE IY+&12,A
```

Adressage indirect

Ce mode d'adressage ressemble à l'adressage indexé, mais la case mémoire est adressée par le contenu des paires de registres HL, BC ou DE.

Format :

```
LD reg,(prs) ou LD (prs),reg
(prs une des paires de registres HL, BC, DE)
```

Chargement de la case mémoire adressée par le contenu du registre prs dans le registre, ou vice versa.

Cette technique d'adressage présente par rapport aux adressages indexé et absolu l'avantage de ne nécessiter que des instructions sur 1 octet. En effet le registre reg et la paire de registres prs sont contenus dans le code d'opération et n'ont donc pas à être fournis comme opérandes. Cette instruction est par conséquent plus rapide alors qu'elle permet également d'accéder à la totalité des 64K.

Exemple :

```
LD B,(HL)            Basic: B=PEEK(HL)
LD (BC),A            Basic: POKE BC,A
```

Nous avons ainsi traité tous les modes d'adressage que l'on rencontre avec les instructions de transfert sur 8 bits. Dans le cours de ce chapitre, nous découvrirons encore quelques autres modes d'adressage et nous étendrons les modes que nous connaissons maintenant à d'autres instructions. Vous trouverez en annexe des tableaux dans lesquels figurent toutes les instructions, triées par tâches (transfert, sauts, etc...) ainsi que les modes d'adressage correspondants.

Ces tableaux vous permettent de retrouver les codes d'opération de toutes les instructions. Nous allons récapituler toutes les instructions de chargement sur 8 bits. Vous trouverez également en annexe une table des abréviations utilisées.

Exemple d'utilisation des listes d'instructions :

SUB (XY+dis) ---> INSTRUCTION

Soustraire une case mémoire adressée de manière indexée du contenu de l'accumulateur et charger le résultat dans l'accumulateur —> EXPLICATION D'INSTRUCTION

A=A-(XY+dis) ---> COMPARAISON

Code d'instruction :	11x11101 &DD	octet 1 code d'opération
	10010110 &96	octet 2 code d'opération
	<--dis-->	octet 3 distance

Flag : S Z V C ---> ETAT DES FLAGS
x x x x

Le 'x' dans le nombre binaire du code d'opération doit être remplacé par 0 lorsque IX est concerné et par 1 lorsque c'est IY qui est concerné.

Liste d'instructions

LD reg,data

Charge la constante data dans le registre reg.

Code d'instruction :	00rrr110	octet 1 code d'opération
	<--co-->	octet 2 constante

rrr correspond à :	A-111	E-011
	B-000	H-100
	C-001	L-101
	D-010	

LD reg,res

Charge le contenu du registre res dans le registre reg.

Code d'instruction :	01rrrsss	octet 1	code d'opération
(sss= registre source)			

LD A,(adr)

Charge le contenu de la case mémoire d'adresse adr dans l'accumulateur

Code d'instruction :	00111010 &3A	octet 1	code d'opération
	<--al-->	octet 2	octet faible adr. abs.
	<--ah-->	octet 2	octet fort adr. abs.

LD (adr),A

Charge le contenu de l'accumulateur dans la case mémoire d'adresse adr

Code d'instruction :	00110010 &32	octet 1	code d'opération
	<--al-->	octet 2	octet faible adr. abs.
	<--ah-->	octet 2	octet fort adr. abs.

LD (HL),data

Charge data dans la case mémoire d'adresse HL.

Code d'instruction :	00110110 &36	octet 1	code d'opération
	<--co-->	octet 2	constante

LD (XY+dis),data

Charge la constante data dans la case mémoire dont l'adresse est fournie par IX ou IY plus dis.

Code d'instruction :	11x11101	octet 1	code d'opération
	00110110 &36	octet 2	code d'opération
	<—dis->	octet 3	distance
	<--co-->	octet 4	constante

LD reg,(XY+dis)

Charge le contenu de la case mémoire adressée par (XY+dis) dans le registre reg.

Code d'instruction :	01x11101 &FD	octet 1	code d'opération
	01rrr110	octet 2	code d'opération
	<--dis->	octet 3	distance

LD (XY+dis),reg

Charge le contenu du registre reg dans la case mémoire d'adresse (XY+dis).

Code d'instruction :	11x11101	octet 1	code d'opération
	01110rrr	octet 2	code d'opération
	<--dis->	octet 3	distance

LD reg,(HL)

Charge le contenu de la case mémoire adressée par HL dans le registre reg.

Code d'instruction :	01rrr110	octet 1	code d'opération

LD (HL),reg

Charge le registre reg dans la case mémoire d'adresse HL.

Code d'instruction : 01110rrr octet 1 code d'opération

LD A,(BC)

Charge le contenu de la case mémoire adressée par la paire de registres BC dans l'accumulateur.

Code d'instruction : 00001010 &0A octet 1 code d'opération

LD A,(DE)

Charge le contenu de la case mémoire adressée par la paire de registres DE dans l'accumulateur.

Code d'instruction : 00011010 &1A octet 1 code d'opération

LD (BC),A

Charge le contenu de l'accumulateur dans la case mémoire adressée par la paire de registres BC.

Code d'instruction : 00000010 &02 octet 1 code d'opération

LD (DE),A

Charge le contenu de l'accumulateur dans la case mémoire adressée par la paire de registres DE.

Code d'instruction : 00010010 &12 octet 1 code d'opération

LD A,I / LD A,R

Charge le contenu du registre d'interruption (I) ou de registre de refresh (R) dans l'accumulateur.

Code d'instruction :	11101101 &ED	octet 1	code d'opération
	0101ss11	octet 2	code d'opération
	ss: I-01		
	R-11		

LD I,A / LD R,A

Charge le contenu de l'accumulateur dans le registre d'interruption ou dans le registre de refresh.

Code d'instruction :	11101101 &ED	octet 1	code d'opération
	0100ss11	octet 2	code d'opération
	ss: I-01		
	R-11		

Vous trouverez en annexe une récapitulation en résumé de ces instructions.

4.2. INSTRUCTIONS DE TRANSFERT SUR 16 BITS

Les instructions de chargement sur 16 bits ont également le format général :

LD objet,source

Mais ce sont 16 bits qui sont chargés de la source dans l'objet. C'est ainsi que ces instructions utilisent les paires de registres BC, DE, HL, SP, IX et IY.

Adressage immédiat

Comme le chargement se fait maintenant avec des registres 16 bits, la constante qui suit le code d'opération doit avoir une taille de 16 bits. Les deux octets suivant le code d'opération contiennent donc l'octet faible et l'octet fort (dans cet ordre!) de la constante. Pour le distinguer de l'adressage immédiat avec des constantes de 1 octet, cette technique d'adressage est appelée adressage immédiat étendu.

Format :

```
LD x,data16
(x: un des registres 16 bits SP, BC, DE, HL, IX, IY)
(data16: une constante de 16 bits)
```

Cette instruction charge la constante data dans le registre x.

Exemple :

```
LD HL,&C000     Basic: HL=&C000
```

Adressage implicite

Pour les instructions 16 bits, il n'y a que trois instructions de ce type, qui concernent toutes le registre SP :

```
LD SP,HL      LD SP,IX    LD SP,IY
```

Ces instructions ont la signification suivante :

Chargement du contenu des registres IIL, IX ou IY dans le pointeur de pile SP.

Comparaison avec le Basic :

```
SP=HL  SP=IX  SP=IY
```

Adressage absolu

Il nous étudier de plus près l'adressage absolu avec les instructions de travail sur 16 bits:

Format :

LD prs,(adr) ou LD (adr),prs
(prs: BL, DE, HL, SP, IX ou IY)

Comme adr indique une adresse et n'adresse donc qu'un octet alors que x est un registre 16 bits, on a adopté la convention suivante:
L'octet faible à l'adresse adr est d'abord chargé dans le registre, puis l'octet fort à l'adresse adr+1.

Par exemple : LD HL,(&AB80) signifie :
registre L = octet faible venant de l'adresse &AB80
registre H = octet fort venant de l'adresse &AB81

Avec l'instruction contraire LD (adr),x, l'octet faible est stocké dans l'adresse adr et l'octet fort dans l'adresse adr+1.

Par exemple : LD (&CB00),IX
adresse &CB00 = octet faible de IX
adresse &CB01 = octet fort de IX

Une instruction de ce type correspond donc à deux instructions de chargement sur 8 bits.

Instruction 16 bits :	Instruction 8 bits:
LD BC,(&FC05) correspond à	LD C,(&FC05) (octet faible)
	LD B,(&FC06) (octet fort)

Comme vous le savez on peut calculer la valeur d'un nombre 16 bits se composant d'un octet fort et d'un faible avec la formule suivante :

Nombre=256*(octet fort)+(octet faible)

Nous obtenons ainsi l'équivalence ci-dessous :

Langage-machine	Basic
LD DE,(&4000)	DE=256*PEEK(&4001)+PEEK(&4000)

En utilisant le systéme hexadécimal, on peut également écrire :

```
DE=VAL("&"+HEX$(PEEK(&4001))+HEX$(PEEK(&4000)))
```

Pour écrire en Basic l'instruction contraire, par exemple LD (&6800),IY deux instructions sont nécessaires:

```
POKE &6800, IY-INT(IY/256)*256   (octet faible)
POKE &6801,    INT(IY/256)       (octet fort)
```

Si vous ne comprenez pas parfaitement ces équivalences entre Basic et langage-machine, consultez à nouveau le chapitre sur la représentation des nombres. Remplacez alors chaque fois DE et IY par des nombres et effectuez vous-même les calculs !

Liste d'instructions

LD prs,data16

Charge la constante data16 dans la paire de registres prs.

Code d'instruction :	00pp0001	octet 1	code d'opération
	<--co-->	octet 2	constante octet faible
	<--co-->	octet 3	constante octet fort

pp correspond à :	BC-00	HL-10
	DE-01	SP-11

LD XY,data16

Charge la constante data16 dans un registre d'index.

Code d'instruction :	11x11101	octet 1	code d'opération
	00100001 &21	octet 2	code d'opération
	<--cl-->	octet 3	constante octet faible
	<--ch-->	octet 4	constante octet fort

LD prs,(adr)

Charge le contenu des adresses adr (octet faible) et adr+1 (octet fort) dans un registre 16 bits.

Code d'instruction :	11101101 &ED	octet 1	code d'opération
	01pp1011	octet 2	code d'opération
	<--al-->	octet 3	adresse octet faible
	<--ah-->	octet 4	adresse octet fort

LD HL,(adr)

Charge le contenu des adresses adr (octet faible) et adr+1 (octet fort) dans le registre HL.

Code d'instruction :	00101010 &2A	octet 1	code d'opération
	<--al-->	octet 2	adresse octet faible
	<--ah-->	octet 3	adresse octet fort

Remarque : Comme cette instruction est fréquemment utilisée, un code sur un octet lui a été attribué alors qu'elle n'est en fait qu'une variante de l'instruction LD prs,(adr) présentée plus haut. L'avantage est que cette instruction est plus courte et plus rapide que le code d'opération normal sur deux octets (&ED,&6B).

LD XY,(adr)

Charge le contenu des adresses adr (octet faible) et adr+1 (octet fort) dans un registre d'index.

Code d'instruction :	11x11101	octet 1	code d'opération
	00101010 &2A	octet 2	code d'opération
	<--al-->	octet 3	adresse octet faible
	<--ah-->	octet 4	adresse octet fort

LD (adr),prs

Charge l'octet faible de prs dans la case mémoire adr et l'octet fort de prs dans l'adresse adr+1.

Code d'instruction :	11101101 &ED	octet 1	code d'opération
	01pp0011	octet 2	code d'opération
	<--al-->	octet 3	adresse octet faible
	<--ah-->	octet 4	adresse octet fort

pp :	BC-00	HL-10
	DE-01	SP-11

LD (adr),HL

Charge l'octet faible de HL (donc L) dans la case mémoire adr et l'octet fort de HL (donc H) dans l'adresse adr+1.

Code d'instruction :	00100010 &22	octet 1	code d'opération
	<--al-->	octet 2	adresse octet faible
	<--ah-->	octet 3	adresse octet fort

Remarque : reportez-vous à la remarque pour l'instruction LD HL,(adr)

LD (adr),XY

Charge l'octet faible du registre d'index dans la case mémoire adr et l'octet fort de ce registre d'index dans la case mémoire adr+1.

Code d'instruction :	11x11101	octet 1	code d'opération
	00100010 &22	octet 2	code d'opération
	<--al-->	octet 3	adresse octet faible
	<--ah-->	octet 4	adresse octet fort

Exercice :

Avant que nous ne poursuivions l'étude des instructions, essayons d'utiliser celles que nous avons étudiées jusqu'ici. Comme vous le savez, la mémoire écran du CPC se trouve à l'adresse &C000. Dans cette zone, chaque groupe de 8 bits (octet) correspond à 8 points se suivant sur l'écran (en MODE 2). L'adresse &C000 est attribuée au premier groupe de 8 points, en commençant par l'angle supérieur gauche de l'écran. Le groupe de 8 points placé immédiatement dessous ce premier groupe figure en &C800, le groupe placé sous ce groupe en &D000, etc... Les groupes de 8 points placés les uns sous les autres sur l'écran sont donc représentés en mémoire par des octets se suivant à intervalle de &800. Entrez par exemple :

```
10 POKE &C000,&FF
20 POKE &C800,&FF
30 POKE &D000,&FF
40 POKE &D800,&FF
50 POKE &E000,&FF
60 POKE &E800,&FF
70 POKE &F000,&FF
80 POKE &F800,&FF
MODE 2
RUN
```

Vous voyez que la case dans l'angle supérieur gauche s'est remplie de la couleur actuelle.

(&FF=&X11111111=8 points mis)

Essayez maintenant de traduire ce programme en langage-machine, en utilisant les instructions que nous avons apprises jusqu'ici. Terminez votre programme en langage-machine par l'instruction RET (&C9).

Discussion de la solution de l'exercice de réalisation d'un programme en langage-machine

Il nous faut d'abord une instruction qui charge une valeur dans une case mémoire (=POKE). Nous pouvons utiliser à cet effet les instructions à adressage indirect, indexé ou absolu (voir définition). Pour traduire précisément notre exemple en Basic, choisissons l'adressage absolu. Nous indiquerons donc chaque fois l'adresse en entier, comme dans le programme Basic. Il est évidemment également possible de stocker l'adresse dans un registre et d'utiliser ensuite soit l'adressage indirect, soit l'adressage indexé.

Exemple :

Basic :	HL=&C000:POKE HL,&FF
Langage-machine :	LD HL,&C000 ou LD (HL),&FF

Comme les instructions sur 16 bits écrivent toujours dans deux cases mémoire consécutives, nous choisirons l'instruction 8 bits:

```
LD (adr),A
```

Avant que cette instruction ne soit exécutée, il faut encore stocker dans l'accumulateur la valeur &FF. On utilise à cet effet l'adressage immédiat :

```
LD A,&FF
```

Notre programme se présente alors ainsi :

```
LD A,&FF
LD (&C000),A
LD (&C800),A
LD (&D000),A
LD (&D800),A
LD (&E000),A
LD (&E800),A
LD (&F000),A
LD (&F800),A
RET
```

Cherchons maintenant quels codes correspondent à ces instructions :

```
LD A,data :      &3E,co
LD (adr),A :     &32,al,ah : octet faible, octet fort
RET :            &C9
```

Nous obtenons ainsi les lignes de DATA que nous pouvons placer dans notre programme de chargement du chapitre 3.1 :

```
10 MEMORY &9FFF
20 FOR i=&A000 TO &A01A
30 READ a
40 POKE i,a
50 NEXT i
60 END
70 DATA &3E,&FF,&32,&00,&C0,&32,&00,&C8
80 DATA &32,&00,&D0,&32,&00,&D8,&32,&00,&E0
90 DATA &32,&00,&E8,&32,&00,&F0,&32,&00,&F8
100 DATA &C9
```

Nous placerons ce programme à l'adresse &A000 (= adresse de départ (V)). Notre programme est long de 27 octets. On peut donc calculer ainsi l'adresse finale (V): &A000+27-1 = &A000+&1A = &A01A. C'est ainsi que nous avons obtenu les valeurs utilisées en ligne 20.

Après que le programme en langage-machine ait été "poké" dans la mémoire (avec RUN), il vous suffit d'entrer >MODE 2< puis >CALL &A000< pour le lancer. Comme vous voyez, la case dans l'angle supérieur gauche de l'écran se colore instantanément. Vous pouvez également entrer ce programme avec le programme de chargement direct. Il vous suffit pour cela de lancer le programme et d'entrer l'adresse de départ &A000. Ensuite viennent les codes (par exemple &3E, &FF, etc...).

C'était donc votre premier programme en langage-machine. Vous pourrez modifier et améliorer ce programme dès que vous aurez appris quelques instructions supplémentaires.

4.3. INSTRUCTIONS DE PILE

Pour comprendre le mode de fonctionnement de la pile, il faut savoir ce qui se déroule à l'intérieur du Z80 lorsqu'on saute à un sous-programme. L'intruction du langage-machine utilisée à cet effet est >CALL adresse<. Le problème principal est que l'unité centrale doit "ranger" quelque part l'adresse de la prochaine instruction, de façon à ce que l'exécution du programme se poursuive après le retour au programme principal (RET).
(Voir figure 5 : Ch. 4.3.).

Comme les registres sont utilisés pour d'autres tâches importantes, les adresses de retour doivent être stockées en dehors de l'unité centrale, donc en RAM. Cette façon de procéder ne permettrait cependant normalement de stocker qu'une seule adresse de retour à la fois. Cela signifie qu'une imbrication de sous-programmes ne serait pas possible. C'est pourquoi une zone de la Ram a été réservée à cet effet. Cette zone est appelée pile (stack). Nous pouvons nous représenter cette pile comme une pile d'assiettes.

Une adresse de retour est notée sur une assiette. L'assiette ainsi dotée d'une adresse est alors placée sur la pile. Il peut y avoir de nombreux appels de sous-programmes qui feront simplement s'élever la pile. Lors d'un retour de sous-programme, on retire de la pile l'assiette placée au sommet de la pile et on saute à l'adresse figurant sur cette assiette. On saute ainsi aux adresses de retour dans l'ordre qui convient, jusqu'à ce que la pile soit éliminée, c'est-à-dire jusqu'à ce qu'on se trouve à nouveau dans le programme principal. L'important est qu'on retire toujours l'assiette qui a été placée en dernier sur la pile (sinon la pile se renverse).

Comme on n'empile pas des assiettes dans l'ordinateur, un registre du Z80 doit être utilisé pour mesurer la hauteur de la pile. Ce registre indique en permanence où se trouve la dernière assiette placée sur la pile. Il est appelé pointeur de pile ou SP (Spack Pointer). Mais en réalité, notre pile est suspendue au plafond, c'est-à-dire que la première assiette est placée à l'adresse la plus élevée de la pile et la dernière assiette à l'adresse la moins élevée. Cette zone que constitue la pile est placée (en descendant) à partir de l'adresse &BFFF.

Le déroulement de l'instruction CALL se présente donc ainsi :

Coupe de la pile :

	'	'
	'	'
	'	'
Pile	&BFF4 :	(entrée précédente)
	&BFF3 :	(entrée précédente)
	&BFF2 :	(entrée précédente)
	&BFF1 :	(dernière entrée)
	&BFF0 :	(place pour de nouvelles entrées) entrée précédente
	&BFEF :	(place pour de nouvelles entrées) entrée précédente
	'	'
	'	'

Pointeur de pile SP :

```
&BFF1
```

Le registre SP indique la dernière entrée dans la pile. Au cours de l'exécution du programme, le processeur rencontre une instruction CALL &B267 à l'adresse &780.

```
&780 CALL &B267
&783 instruction suivante
```

Après lecture de l'instruction, le PC se trouve sur &783. C'est l'adresse de retour qu'il s'agit de ranger. L'adresse est placée sur la pile dans le format octet faible et octet fort. SP est alors diminué, l'octet fort est stocké dans l'adresse SP, SP est à nouveau diminué et l'octet faible est stocké dans la nouvelle adresse SP. L'adresse fournie pour le début du sous-programme est alors chargée dans le PC, ce qui signifie que l'exécution du programme se poursuit à partir de cette adresse. On obtient donc la situation suivante :

```
          ,          ,
          ,          ,
          ,          ,
Pile :  &BFF0      &07
        &BFEF      &83          : (dernière entrée)
          ,          ,
          ,          ,
SP : &BFEF
```

Comme vous le voyez, SP indique à nouveau l'emplacement de la dernière entrée. Lors de l'exécution de l'instruction RET, la procédure se déroule dans le sens inverse :

L'octet figurant dans la case mémoire indiquée par le pointeur de pile SP est chargé, comme octet faible, dans le PC. Le pointeur de pile SP est augmenté de 1 et l'octet fort de l'adresse de retour est chargé dans le PC. SP est alors à nouveau augmenté de 1, c'est-à-dire qu'il indique l'adresse de retour suivante sur la pile.

L'exécution de programme se poursuit alors à l'adresse PC, c'est-à-dire à l'adresse de retour correcte.

```
        ,        ,
        ,        ,
Pile :  &BFF1    ...          SP : &BFF1
        &BFF0    &07
        &BFEF    &83
        ,        ,
```

Les procédures décrites ci-dessus se déroulent automatiquement dans le Z80, dès qu'un CALL ou un RET se produisent. Cela garantit que l'ordre dans la pile soit toujours correct et que le SP indique le bon emplacement de la pile. Si vous modifiez le pointeur de pile SP par programmation, l'ordre peut être aisément détruit ce qui "plante" l'ordinateur. Maniez donc les instructions LD SP,x avec prudence.

Il est possible de placer ou de retirer également des données sur la pile. C'est à cela que servent les instructions :

PUSH (placer sur la pile)
et
POP (retirer de la pile)

PUSH fonctionne de façon analogue à l'instruction CALL. Les données à stocker sont décrites sur la pile après que le SP ait été diminué. Avec POP, les données sont lues et le SP est automatiquement augmenté. Toutes les opérations nécessaires sont ici aussi prises en charge par l'unité centrale. PUSH et POP permettent de placer sur la pile tous les registres 16 bits, hormis le registre SP lui-même.

Format :

```
PUSH x              POP x
(x:AF, BC, DE, HL, IX, IY)
```

Comme l'accumulateur est un registre 8 bits et qu'il est également intéresant de placer le registre F (flags) sur la pile, A et F sont traités ensemble.

La technique du stockage provisoire sur la pile est intéressante à utiliser lorsque les registres ne suffisent plus au stockage des données.

Exemple :

HL contient le premier opérande de l'addition
BC contient le second opérande de l'addition

Un sous-programme est alors appelé qui additionnera HL et BC. Le résultat de l'addition est stocké dans HL. Si on a ensuite encore besoin du premier opérande, il faudra qu'il ait été placé à temps sur la pile.

```
                LD HL, opérande un
                LD BC, opérande deux
                PUSH HL
                CALL addition
                  .
                  .
Addition          .
                POP HL
                  .
                  .
```

Si on a besoin de ce premier opérande, on peut le retirer de la pile avec POP HL.

Il faut faire attention à ce que l'instruction POP correspondant à un PUSH figure toujours dans le même sous-programme. Sinon les données stockées avec PUSH seront utilisées comme adresse de retour par la prochaine instruction RET. Ceci conduira selon toute probabilité à un "plantage" de l'ordinateur. Les instructions PUSH et POP n'ont pas d'équivalent en Basic. Elles peuvent être écrites en Basic de la façon suivante :

Exemple Basic :

```
PUSH AF          Basic :   POKE SP-1,A:(octet fort)
                           POKE SP-2,F
                           SP=SP-2

POP BC           Basic :   BC=PEEK(SP)+256*PEEK(SP+1)
                           SP=SP+2
```

Comme PUSH et POP utilisent SP comme pointeur d'adresse, elles comptent au nombre des instructions avec adressage indirect.

Exemple :

```
PUSH HL          SP=&BE05
                 HL=&1234
```

Après exécution : case mémoire &BE04:&12
case mémoire &BE03:&34

```
SP = &BE03
HL = &1234
```

Exemple :

```
POP HL           SP=&BE03
                 HL=&FFFF
```

Après exécution :

```
SP = &BE05
HL = &1234
```

Liste d'instructions

PUSH par,x

Sauver le registre par sur la pile (avec modification automatique de SP).

Code d'instruction :	11pp0101	Octet 1 Code d'opération

pp: BC-00 HL-10
DE-01 AF-11

PUSH XY

Sauver le registre d'index sur la pile (avec modification automatique de SP).

Code d'instruction :	11x11101	Octet 1 Code d'opération
	11100101 &E5	Octet 2 Code d'opération

POP par,x

Retirer deux octets de la pile et les charger dans le registre par (avec modification automatique de SP).

Code d'instruction :	11pp0001	Octet 1 Code d'opération

POP XY

Retirer deux octets de la pile et les charger dans le registre d'index (avec modification automatique de SP).

Code d'instruction :	11x11101	Octet 1 Code d'opération
	11100001 &E1	Octet 2 Code d'opération

4.4. INSTRUCTIONS D'ECHANGE

Le Z80 dispose, outre les instructions de transfert simple des données (LD), d'instructions qui échangent entre eux les contenus de deux emplacements. Ces instructions sont représentées par EX (exchange : échanger).

Une instruction de ce type comme EX DE,HL échange par exemple le contenu du registre DE avec celui du registre HL. L'instruction EX avec adressage indirect échange le contenu des registres HL, IX ou IY avec l'élément le plus élevé de la pile (SP n'est pas modifié).

Format :

```
EX (SP),x
x: HL, IX ou IY
```

Il y a d'autre part des instructions d'échange qui réalisent l'échange avec le contenu du deuxième jeu de registres. Comme nous l'avons déjà indiqué, à chaque registre A, BC, DE, HL et F correspond un deuxième registre A', BC', DE', HL' et F'. On travaille cependant toujours avec le premier jeu de registres (A-F). Mais on peut si besoin est échanger entre eux les contenus des deux jeux de registres.

L'instruction EX AF,AF' échange les contenus de l'accumulateur et du registre de flags avec les registres correspondants A' et F'. L'instruction EXX échange les autres paires de registres BC, DE et HL contre leurs doubles respectifs BC', DE' et HL'.

Ces instructions sont à adressage implicite.

Exemple :

```
EX DE,HL        Basic :  PROV = HL : HL = DE : DE = PROV

EX (SP),HL      Basic :  PROV = HL : HL = 256*PEEK(SP+1)+PEEK(SP):
                         POKE SP+1,INT(PROV/256) : POKE SP,
                         PROV - INT(PROV/256)*256
```

Liste d'instructions

EX DE,HL

Echanger les contenus des registres DC et HL

Code d'instruction : 11101011 &EB Octet 1 Code d'opération

EX (SP),HL

Echanger le contenu du registre HL avec celui de l'élément le plus élevé de la pile.

Code d'instruction : 11100011 &E3 Octet 1 Code d'opération

EX (SP),XY

Echanger le contenu du registre d'index avec celui de l'élément le plus élevé de la pile.

Code d'instruction :	11X11101	Octet 1	Code d'opération
	11100011 &E3	Octet 2	Code d'opération

EX AF,AF'

Echanger le contenu du registre AF avec celui de son double AF'.

Code d'instruction : 00001000 &08 Octet 1 Code d'opération

EXX

Echanger le contenu des registres BC, DE, HL avec celui de leur double BC', DE', HL'.

Code d'instruction : 11011001 &D9 Octet 1 Code d'opération

4.5. INSTRUCTIONS DE TRANSFERT ET DE RECHERCHE DE BLOC

Les instructions de transfert de bloc ne transfèrent pas, comme LD, uniquement un ou deux octets, mais un bloc entier de données. Ces instructions sont une particularité du Z80. Normalement, ces instructions n'existent pas sur les micro-processeurs car elles sont assez coûteuses à réaliser pour le constructeur. Mais pour le programmeur, ces instructions sont très utiles. Elles augmentent en effet la puissance d'un programme.
Un bloc de données est défini avec les indications suivantes :

- L'adresse de début ou l'adresse de fin du bloc qui est stockée dans hl

- La longueur du bloc en octets qui est stockée dans BC (Byte Counter)

Avec ces deux données, il est possible de définir des blocs d'une longueur pouvant aller jusqu'à 64 K et qui peuvent commencer en n'importe quel endroit de la mémoire. Comme il s'agit de transférer ce bloc, il faut encore indiquer, dans DE, l'adresse de début ou l'adresse de fin du bloc objet. Une fois que ces données ont été placées dans les registres, le transfert de bloc proprement dit peut être réalisé. Il y a quatre instructions de transfert de bloc :

LDD, LDDR, LDI, LDIR

Chaque instruction de transfert de bloc décrémente (diminue la valeur de) le compteur BC après chaque transfert d'un octet. Deux de ces instructions incrémentent ensuite (augmentent la valeur de) les pointeurs HL et DE qui sont alors dirigés sur les adresses source et objet du prochain octet à transférer.

Avec LDD et CDDR au contraire, les pointeurs sont décrémentés, c'est-à-dire que le bloc est transféré en commençant "par le haut". Pour ces deux instructions, ce sont les adresses finales source et objet du bloc qui doivent être chargées dans HL et DE.

Le R à la fin de ces instructions signifie Répète. L'exécution de ces instructions se répète jusqu'à ce que BC soit égal à 0, donc jusqu'à ce que la totalité du bloc ait été transmise. Voici maintenant une description de chaque instruction :

LDI : charge (LoaD) et Incrémente

Cette instruction transfère un octet de l'adresse HL à l'adresse DE. BC est ensuite décrémenté. Les pointeurs d'adresse HL et DE sont incrémentés, de façon à ce que tout soit prêt pour une éventuelle continuation du transfert.
Pour cela, il suffit d'appeler à nouveau cette instruction.

LDIR : charge (LoaD), Incrémente et Répète

La procédure de transfert se déroule comme pour LDI. Le PC est ensuite automatiquement dirigé à nouveau sur cette instruction qui est donc à nouveau exécutée, jusqu'à ce que BC = 0. Le programme passe alors seulement à l'exécution de l'instruction suivante.

LDD : charge (Loa D) et Décrémente

Cette instruction est semblable à LDI, mais le transfert de bloc s'opère en commençant par la fin du bloc, c'est-à-dire que HL et DE sont décrémentés. Cette différence est importante lorsque les blocs source et objet se recoupent. Si l'on n'utilise pas en effet la bonne instruction, des données du bloc source risquent en effet d'être effacées avant d'être transférées.
(voir figure 6 : Ch. 4.5.)

LDDR : charge (LoaD), Décrémente et Répète

La procédure de transfert se déroule comme pour LDD mais, comme pour LDIR, cette instruction est à nouveau exécutée, jusqu'à ce que le bloc entier ait été transmis.

Exemple :

```
LDIR            BASIC : 10 POKE DE,PEEK(HL)
                        20 HL = HL+1
                        30 DE = DE+1
                        40 BC = BC+1
                        50 IF BC<>0 THEN 10

LDD             BASIC : POKE DE,PEEK(HL) :
                        DE = DE-1 : HL = HL-1 : BC = BC-1
```

Essayez de trouver par vous-même les équivalents Basic de LDDR et LDI.

Le nombre d'instructions utilisées en Basic vous montre qu'il s'agit d'instructions très puissantes.

Modification des flags : lorsque BC = 0 après exécution d'une de ces instructions, P/V = 0

Les instructions avec répétition, LDDR et LDIR mettent toujours P/V à 0.

Instructions de recherche de bloc

Les instructions de recherche de bloc permettent de rechercher un octet déterminé dans un bloc de données. L'octet recherché est auparavant placé dans l'accumulateur. Si l'instruction rencontre au cours de la recherche un octet identique au contenu de l'accumulateur, le flag Z est mis et les instructions avec répétition ne sont plus répetées. Les registres sont utilisés comme pour les instructions de transfert de bloc.

HL- Adresse de début ou de fin de bloc
BC- Byte Counter : longueur du bloc
DE- n'a pas de fonction.
L'accumulateur contient l'octet à rechercher

CPIR compare lors de chaque parcours le contenu de la case mémoire HL avec le contenu de l'accumulateur. HL est alors incrémenté et BC est décrémenté. Si BC = 0, le flag P/V est mis sur 0, sinon sur 1. Si le résultat de la comparaison de 1 et (HL) est positif, le flag Z est mis, sinon il est annulé.

Le flag S correspond, comme pour CP au 7ème bit du résultat de la soustraction A - (HL). Le carry n'est pas modifié. Quatre instructions de rechercher de bloc sont possibles :

CPI, CPIR, CPD, CPR

Leur mode de travail correspond aux modes de travail des instructions de transfert correspondantes.

Toutes les instructions de bloc sont des instructions sur 2 octets dont le premier code d'opération est &ED. De même que les instructions de transfert de bloc, les instructions de recherche sont des instructions qui rendent dans de nombreux domaines la programmation plus aisée et plus rapide.

Nous représenterons dorénavant la fonction de chaque instruction de manière symbolique. Nous utiliserons les conventions suivantes :

= pour : transférer les données de ... à. (comme en Basic)

() pour : charger le contenu de la case mémoire qui est adressée par le contenu de l'adresse ou registre entre parenthèses. (comme PEEK).

Liste d'instructions

LDI

Transfert de données avec incrémentation

(DE) = (HL), DE = DE+1, HL = HL+1, BC = BC-1

Code d'instruction :	11101101 &ED	Octet 1	Code d'opération
	10100000 &AO	Octet 2	Code d'opération

Flags : P/V mis lorsque BC = 0, sinon annulé.

LDIR

Transfert de données avec incrémentation et répétition

(DE) = (HL), DE = DE+1, HL = HL+1, BC = BC-1, répéter jusqu'à ce que BC = 0

Code d'instruction :	11101101 &ED	Octet 1	Code d'opération
	10110000 &BO	Octet 2	Code d'opération

Flags : P/V = 1

LDD

Transfert de données avec décrémentation

(DE) = (HL), DE = DE-1, HL = HL-1, BC = BC-1

Code d'instruction :	11101101 &ED	Octet 1	Code d'opération
	10101000 &A8	Octet 2	Code d'opération

Flags : P/V mis lorsque BC = 0, sinon annulé.

LDDR

Transfert de données avec décrémentation et répétition

(DE) = (HL), DE = DE-1, HL = HL-1, BC = BC-1, répéter jusqu'à ce que BC = 0

Code d'instruction :	11101101 &ED	Octet 1	Code d'opération
	10111000 &B8	Octet 2	Code d'opération

CPI

Recherche dans un bloc avec incrémentation

A = (HL), HL = HL+1, BC = BC-1

Code d'instruction :	11101101 &ED	Octet 1	Code d'opération
	10100001 &A1	Octet 2	Code d'opération

Flags : P/V mis lorsque BC-1<>0, sinon annulé.
Z mis lorsque A = (HL), sinon annulé.
S correspond au bit 7 de A - (HL).

CPIR

Recherche dans un bloc avec incrémentation et répétition

A = (HL), HL = HL+1, BC = BC-1

Code d'instruction :	11101101 &ED	Octet 1	Code d'opération
	10110001 &B1	Octet 2	Code d'opération

Flags : P/V mis lorsque BC-1<>0, sinon annulé.
Z mis lorsque A = (HL), sinon annulé.
S correspond au bit 7 de A - (HL).

CPD

Recherche dans un bloc avec décrémentation

A = (HL), HL = HL-1, BC = BC-1

Code d'instruction :	11101101	&ED	Octet 1	Code d'opération
	10101001	&A9	Octet 2	Code d'opération

Flags : P/V mis lorsuqe BC-1<>0, sinon annulé.
Z mis lorsque A = (HL), sinon annulé.
S correspond au bit 7 de A - (HL).

CPDR

Recherche dans un bloc avec décrémentation et répétition

A = (HL), HL = HL-1, BC = BC-1

Code d'instruction :	11101101	&ED	Octet 1	Code d'opération
	10111001	&B9	Octet 2	Code d'opération

Flags : P/V mis lorsuqe BC-1<>0, sinon annulé.
Z mis lorsque A = (HL), sinon annulé.
S correspond au bit 7 de A - (HL).

Exercice :

Pour bien comprendre l'instruction LDDR, nous allons l'essayer immédiatement. Nous allons décaler le contenu de l'écran d'un caractère vers la droite. Comme un octet correspond exactement à la largeur d'un caractère, il nous faut donc décaler le bloc &C000 à &FFFF d'un octet vers le haut.

Ecrivez donc au moyen des instructions de transfert de bloc un programme en langage-machine qui exécute cette tâche.

Solution

Analysons tout d'abord notre problème :

Le bloc source figure dans la zone &C000-&FFFE.

Il nous faut décaler ce bloc d'un octet vers le haut, donc vers la zone &C001-&FFFF. Les deux blocs se chevauchent de façon évidente. Comme l'adresse finale du bloc source &FFFE se chevauche avec le bloc objet, c'est l'instruction LDDR qui doit être choisie.

Calculons maintenant le contenu des registres HL, DE, BC. HL doit contenir l'adresse finale du bloc source, donc &FFFE. BC contient le nombre d'octets à décaler. Ce nombre est de &4000-1 (la zone de l'écran de &C000-&FFFF a une taille de 84000 octets). Donc BC=&3FFF. DE contient l'adresse finale du bloc objet, donc &FFFF.

Nous obtenons ainsi le programme en langage-machine suivant :

```
LD HL,&FFFE
LD DE,&FFFF
LD BC,&3FFF
LDDR
RET
```

Après traduction de ce programme, nous obtenons pour le programme de chargement Basic les lignes DATA suivantes :

```
DATA &21,&FE,&FF,&11,&FF,&FF
DATA &01,&FF,&3F,&ED,&B8
DATA &C9
```

(l'adresse de début est &A000 et l'adresse finale &A00B).

Entrez maintenant >MODE 2<, chargez le programme en langage-machine avec >RUN< et lancez-le avec >CALL adresse<.

Notre programme pêche par une petite inélégance :

La case supérieure gauche comporte un point en haut. Pour que celui-ci disparaisse, il nous faut charger 0 dans la case mémoire correspondante &C000.

```
LD A,00
LD (&C000),A
Code : &3E,&00,&32,&00,&C0
```

Il nous faut ajouter ces instructions après l'instruction LDDR. La dernière ligne de DATA devient alors :

```
DATA &3E,&00,&32,&00,&C0,&C9
```

(l'adresse finale devient &A010).

Après avoir testé ce programme, entrez la ligne suivante :

```
FOR I=1 TO 80:CALL &A000:NEXT
```

Le résultat de cette instruction est que l'écran est décalé d'une ligne vers le bas. Le temps nécessaire pour l'exécution est cependant relativement important car les 16 k de l'écran doivent être décalés 80 fois. En Basic, ce décalage nécessiterait environ une heure. Si au lieu de cela nous décalons directement le bloc de l'écran de 80 caractères, le temps d'exécution serait 80 fois moindre. Il nous faut pour cela modifier le contenu des registres dans notre programme en langage-machine :

HL doit contenir &FFFF-80 au lieu de &FFFF-1, donc &FFAF. DE contient toujours &FFFF.

Le nombre d'octets à décaler est de &4000-80=&3FB0.

Modifiez comme il convient les lignes de DATA et notre programme décalera l'écran d'une ligne vers le bas. Malheureusement, les 80 premiers octets de la mémoire écran ont toujours leur ancien contenu. Il faut donc les annuler ! Pour cela aussi nous allons utiliser l'instruction de transfert de bloc.

Pour que cette instruction efface une zone, nous devons l'utiliser volontairement de façon incorrecte : nous stockons tout d'abord l'octet nul à l'adresse &C000. (LD (&C000),0). Nous décalons ensuite le bloc de &C000 à &C000+80=&C050 vers &C001.

Comme les zones se chevauchent à l'adresse finale du bloc source, nous devrions normalement utiliser LDDR.

Mais si nous prenons LDIR, HL=&C000, DE=&C001, BC=&4F, la prochaine case mémoire devant être transférée sera toujours effacée avec la valeur de la case mémoire qui vient d'être transférée. Comme &C000 vaut 0, tous les octets de ce bloc recevront la valeur 0 ! !

Le programme complet à maintenant la forme suivante :

Adresse/	Code/	N° ligne Basic/	Instructions assembleur
A000	21AFFF	10	LD HL,&FFAF
A003	11FFFF	20	LD DE,&FFFF
A006	01B03F	30	LD BC,&EFBO
A009	EDB8	40	LDDR
A00B	3200C0	50	LD (&C000),A
A00E	2100C0	60	LD HL,&C000
A011	1101C0	70	LD DE,&C001
A014	014F00	80	LD BC,&4F
A017	EDB0	90	LDIR
A019	C9	100	RET

Explication du listing assembleur :

Les adresses sont calculées en permanence de façon à tenir compte du nombre d'octets dans le code. Comme un octet est toujours indiqué par deux chiffres hexadécimaux, nous obtenons bien l'écart entre A000 et A003 qui peut surprendre de prime abord.

Le code se compose ici de 3 octets : &21, &00, &C0. Comme chaque octet augmente l'adresse de 1, l'adresse de début de l'instruction suivante est donc A003 (A000+3 = A003). Le nombre de codes permet de déterminer aisément la longueur de l'instruction.

Les instructions assembleur sont placées à la suite des codes. Nous expliquerons plus tard leur fonction.

Si vous avez provoqué un "scrolling" (décalage) de l'écran en travaillant sur votre ordinateur, des irrégularités troubleront le déroulement du programme en langage-machine.

Ce phénomène ne peut cependant pas apparaître si vous videz l'écran avec l'instruction MODE 2 avant d'appeler le programme. Essayez également la ligne suivante :

```
FOR I = 1 TO 26 : CALL &A000 : NEXT
```

Cette instruction devrait normalement vider les 25 lignes de l'écran. Les lignes qui disparaissent du bord inférieur de l'écran réapparaissent cependant dans le bord supérieur, au milieu de la ligne.

Cela vient d'une part de la structure de la mémoire écran et d'autre part du fait que le scrolling intégré fonctionne différemment. Mais nous nous attaquerons de nouveau à ce problème lorsque nous aurons appris quelques nouvelles instructions.

Faites encore quelques expériences avec les instructions de transfert de bloc : utilisez différentes valeurs pour HL, DE et BC.

Mais faites attention à ce que le bloc objet ne sorte jamais de la zone &C000-&FFFF. Cela risquerait en effet de "planter" l'ordinateur car des routines système seraient ainsi effacées.

Vous pouvez également essayer la chose suivante :

HL = &C000, DE = &FFFF, BC = &3FFE

4.6. INSTRUCTIONS ARITHMETIQUES

Les premiers ordinateurs digitaux apparus dans les années 50 étaient essentiellement conçus comme des machines à calculer. Bien que les ordinateurs de l'époque aient peu de points communs avec ceux d'aujourd'hui, les instructions arithmétiques restent semblables. Il y a deux opérations arithmétiques de base, l'addition et la soustraction auxquelles correspondent les instructions du langage-machine ADD et SUB. Comme l'ordinateur calcule en binaire (système numérique de base 2), voyons tout d'abord comment ces opérations sont exécutées dans ce système numérique.

Addition :

En système décimal, on additionne deux chiffres placés l'un au-dessus de l'autre. Le chiffre des unités du résultat est noté et si un chiffre des dizaines (la retenue) apparaît, il est noté pour être pris en compte lors de l'addition des chiffres suivants.

Exemple:

```
       3573
  +    7154      ( * Pour l'addition, vous devriez noter ici un 1.
       -----       Ce chiffre correspond à la retenue.)
      10727
       * *
```

Une retenue se produit dès que la somme de deux chiffres est plus grande que 9 (10-1). En système binaire, une retenue se produit dès que la somme de deux chiffres est plus grande que 1 (2-1).

Règles :

```
0 + 1 = 1
1 + 0 = 1
0 + 0 = 0
1 + 1 = 0      <---( pour ce dernier calcul, il faut bien sûr
                     noter une retenue)
```

Application :

```
    1 0 0 1 0 1 1 0 = &96  =  150
  + 0 0 1 1 1 0 0 1 = &39  =   57
    -------------------------------
    1 1 0 0 1 1 1 1 = &CF  =  207
      * *
```

(* signifie : retenue de 1 ! !)

En hexadécimal la règle devient :

Une retenue se produit dès que la somme de deux chiffres est plus grande que 15 (16-1).

```
    0 0 1 0 1 1 1 0 = &2E  =   46
  + 0 0 0 1 0 1 1 1 = &17  =   23
    -------------------------------
    0 1 0 0 0 1 0 1 = &45  =   69
```

&E+&7 = 14+7 = 21 = &15

donc : noter 5, retenir 1 !

Par ailleurs nous avons, dans l'exemple ci-dessus, un cas supplémentaire est venu s'ajouter aux précédents en ce qui concerne l'addition binaire :

```
     11
  +  11
    -----
    110
```

Pour le deuxième chiffre, c'est la règle suivante qui s'applique :

1 + 1 + 1 = 1, et je retiens 1 !

Exercices :

1)
```
  1 0 1 0 1 1 1 0 = & ? = ?
+ 0 0 1 0 1 1 1 1 = & ? = ?
---------------------------
         ?        = & ? = ?
```

2)
```
  0 0 1 1 1 1 1 1 = & ? = ?
+ 0 0 1 0 1 1 1 1 = & ? = ?
---------------------------
         ?        = & ? = ?
```

3)
```
  1 1 1 1 1 1 1 1 = & ? = ?
+ 1 1 0 0 1 0 1 0 = & ? = ?
---------------------------
         ?        = & ? = ?
```

Solutions :

1)
```
  1 0 1 0 1 1 1 0 = &AE  = 174
+ 0 0 1 0 1 1 1 1 = &2F  =  47
-------------------------------
  1 1 0 1 1 1 0 1 = &DD  = 221
```

2)
```
  0 0 1 1 1 1 1 1 = &3F  =  63
+ 0 0 1 0 1 1 1 1 = &2F  = 157
-------------------------------
  1 1 0 1 1 1 0 0 = &DC  = 220
```

3)
```
  1 1 1 1 1 1 1 1 = &FF  = 255
+ 1 1 0 0 1 0 1 0 = &CA  = 202
-------------------------------
1 1 1 0 0 1 0 1 1 =&1C9  = 457
```

Pour le point 3) : dans cette addition, une retenue se produit du chiffre 8 (bit 7) au chiffre 9 (bit 8). Un octet ne comporte cependant que 8 chiffres (8 bits). Ce bit de retenue (en anglais CARRY) est stocké pour cette raison dans le bit 0 du registre flags. En principe, il est évidemment possible d'additionner même des nombres comportant un nombre encore plus important de chiffres. Mais avec un ordinateur, il faut procéder autrement.

Soustraction

La soustraction en binaire est semblable à la soustraction en décimal. Les règles suivantes s'appliquent :

```
0 - 1 = 1  je retiens 1
1 - 0 = 1
0 - 0 = 0
1 - 1 = 0
```

Voici un exemple :

```
  01101110  =  &6E  =  110
- 00110101  =  &35  =   53
  ---------    ----    ----
  00111001     &39      57
    **   *
```

Ceci nous indique les règles pour le calcul avec retenue:

```
1 - (1+1) = 1  je retiens 1
0 - (1+1) = 0  je retiens 1
```

Exercices :

Faites les exercices pour l'addition en effectuant des soustractions. Contrôlez vous-même la justesse de vos résultats en les convertissant en système décimal.

Pour le point 2) : le résultat de la soustraction est ici négatif. Le résultat correct serait 63-157=-84. En binaire nous obtenons:

```
  00111111
- 10011101
  --------
 110100010 = &1A2
```

Il est clair que ce résultat est mauvais. La soustraction en binaire sur ordinateur pose le problème de la représentation des nombres négatifs. On a dans ce domaine adopté la convention suivante :

Le bit 7 d'un nombre binaire est utilisé comme bit-signe. 0 signifie qu'il s'agit d'un nombre positif, 1 qu'il s'agit d'un nombre négatif. La zone des valeurs numériques pouvant être représentées par un octet a donc ainsi pour limites -128 et +127. La soustraction de nombres binaires revient ainsi à additionner des nombres signés : 5 - 2 = 5 + (-2) !

La représentation signée, telle qu'elle est utilisée pour la soustraction, est appelée complément à deux.

Qu'est-ce que le complément à deux ?

Dans la représentation du complément à deux, les nombres positifs continuent à être représentés comme précédemment, par exemple :

5=&X0000101, 126=&X01111110.

Pour représenter un nombre négatif, on calcule tout d'abord son complément. Le complément d'un nombre binaire est un nombre binaire dont l'état de tous les bits est contraire à l'état des bits du premier nombre. A 0 correspond donc 1 et à 1, 0. Le nombre binaire ainsi obtenu est appelé complément à 1 ou complément simple.

Exemple :

```
Nombre:       7=&X00000111
Complément:     &X11111000
```

Pour obtenir le complément à deux du nombre, il faut ajouter 1 au complément simple.

Exemple :

```
Complément :          &X11111000
plus 1            +            1
                  ------------------
Complément à 2        &X11111001
```

Nous obtenons ainsi la représentation de -7 en complément à 2.

Le complément à deux se forme donc de la façon suivante:

- un nombre positif reste inchangé
- on forme le complément d'un nombre négatif et on ajoute 1

Représentation en complément à deux :

Décimal	Complément à deux
+ 127	01111111
+ 126	01111110
+ 125	01111101
.	.
.	.
.	.
+ 2	00000010
+ 1	00000001
0	00000000
- 1	11111111
- 2	11111110
- 3	11111101
.	.
.	.
.	.
- 126	10000010
- 127	10000001
- 128	10000000

Pour obtenir la valeur d'un nombre négatif représenté en complément à deux, on forme à nouveau le complément à deux de la représentation en complément à deux de ce nombre négatif.

Exemple :

```
  &X00000111  Complément
+          1  plus 1
----------------
  &X00001000

              &X00001000=8
```

La valeur de &X11111000 est donc -8 !

Si on procède deux fois successives à la formation du complément ê deux, on retombe sur le nombre que l'on avait au départ.

Le Z80 dispose d'instructions de conversion du contenu de l'accumulateur en son complément (CPL) et en son complément à deux (NEG). Nous allons tout d'abord reproduire la fonction de ces instructions en Basic :

Considérons tout d'abord la formation du complément :

Mettons que A contienne un nombre entre 0 et 255 (1 octet). L'instruction BIN$ convertit tout nombre en une chaîne alphanumérique qui correspond à l'expression binaire de ce nombre ! Nous formerons ensuite, bit pour bit, le complément de cette chaîne.

```
10 A=&X11011
20 abin$=BIN$(a,8)
30 PRINT "Nombre binaire":";abin$
40 FOR i=0 TO 7
50 bit$=MID$(abin$,8-i,1):REM bit No i
60 IF bit$="1" THEN bit$="0" ELSE bit$="1"
70 acpl$=bit$+ acpl$:REM acpl$ est le ComPLement $ de a
80 NEXT
```

```
90 PRINT "Complement:";acpl$
100 A=VAL("&X"+acpl$)
```

La ligne 50 extrait chaque fois l' i-iéme bit de abin$. La ligne 60 forme le complément de ce bit (0=>1 et 1=>0). Les compléments des différents bits sont regroupés dans acpl$. Ce programme est toutefois très lent. L'instruction XOR en Basic forme le complément plus rapidement. Nous ne vous donnons ici que le programme utilisant cette instruction logique dont le fonctionnement sera expliqué au chapitre suivant.

```
10 A=&X11011
20 abin$=BIN$(a,8)
30 PRINT "Nombre binaire":";abin$
40 a=a XOR 255
50 acpl$=BIN$(a,8)
60 PRINT "Complement:";acpl$
```

C'est la ligne 40 qui forme le complément.

L'instruction NEG convertit un nombre positif en un nombre négatif en représentation en complément à deux. En Basic, cela se présente ainsi :

```
10 A=&X11011
20 abin$=BIN$(a,8)
30 PRINT "Nombre binaire":";abin$
40 a=a XOR 255
45 a=a+1
50 acpl$=BIN$(a,8)
60 PRINT "Complement a deux:";acpl$
```

Ajoutez encore la ligne suivante :

```
100 GOTO 40
```

Vous constatez ainsi que la répétition de l'opération de formation du complément à deux ramène toujours au point de départ.

Avec la représentation en complément à deux, la soustraction de deux nombres peut maintenant être considérée comme l'addition d'un de ces deux nombres au complément à deux de l'autre nombre. D'autre part, le résultat d'une soustraction est considéré comme étant un nombre négatif (en représentation en complément à deux) lorsque le bit 7 (bit-signe) est mis.

Exemple :

```
120 - 63 = 57
120 = &X01111000
 63 = &X00111111
```

Le complément à deux de 63 est &X11000001

Additionnons maintenant :

```
       01111000   = 120
+      11000001   = complément à deux de 63
-----------------
      100111001
```

Négligeons pour le moment la retenue du bit 7 dans le bit 8 (carry). le résultat est correct : &X00111001 = 57

Le bit 7 n'est pas mis, donc le résultat est positif. Le Carry ne devrait donc pas normalement être mis.

Mais comme nous travaillons avec le complément à deux, on peut dire qu'on forme également le complément du Carry. Dans le cas présent, il n'est pas nécessaire de tenir compte du Carry. Le résultat est correct malgré tout. Un examen plus approfondi de l'arithmétique des nombres signés montre que plusieurs cas spéciaux doivent être pris en compte. Le rôle joué par les flags ensemble est important à cet égard.

Exercice :

Calculez le complément à deux de :

1) - 60
2) - 120
3) + 5
4) - 6

Solutions :

1) &X11000100(=196)
2) &X10001000(=136)
3) &X00000101(=5)
4) &X11111010(=250)

Instructions arithmétiques et de comptage sur 8 bits

Il y a deux instructions pour l'addition et pour la soustraction :

ADD;ADC et SUB;SBC

Pour les instructions dont le nom se termine par C (Carry), le flag carry est pris en compte pour l'opération. En effet, lorsque vous utilisez une de ces deux instructions, le bit 0 du registre F (le Carry!) est additionné ou soustrait.

Les opérandes de ces instructions suivent le format :

A,x où x est mis pour reg, data, (HL) ou (XY+dis).

Nous obtenons ainsi les types d'instruction suivants :

A,reg	- implicite
A,data	- immédiat
A,(HL)	- indirect
A,(XY+dis)	- indexé

Pour l'instruction SUB, seuls reg, data, (HL) ou (XY+dis) sont indiqués comme opérandes. "A" est négligé car toutes les instructions de ce type se rapportent à l'accumulateur.
Ces instructions sont des opérations sur 8 bits. Le Z80 dispose en outre d'instructions arithmétiques sur 16 bits.

Lors de l'exécution des instructions de traitement de données, les flags sont modifiés :

Flag Carry

Le flag Carry est mis lorsque se produit une retenue du bit 7 sur le bit 8. Comme un octet ne se compose que des huits bits 0 à 7, cette retenue est sauvegardée dans le flag C. Si aucune retenue ne se produit, le flag Carry est annulé.

Flags N et H

Ces flags sont modifiés mais ils n'ont pour nous aucune signification pour le moment.

Flag de dépassement P/V

Un dépassement se définit ainsi :

- lorsque une retenue interne du bit 6 au bit 7 se produit, sans qu'il n'y ait de retenue externe du bit 7 au bit 8 (indiquée par le Carry)

- lorsqu'il n'y a pas de telle retenue interne, mais qu'il y a une retenue externe

Nous ne chercherons pas à comprendre comment on en arrive à ces définitions. Ce qui importe pour nous, c'est que ce flag est mis lorsqu'au cours d'une opération arithmétique le signe du résultat (bit 7) a été modifié de façon injustifiée. Le flag V est mis lorsqu'une retenue se produit, sinon il est annulé.

Flag Zéro

Ce flag est mis lorsque le résultat d'une opération a été 0, sinon il est annulé.

Flag Signe

Ce flag correspond au bit 7 du résultat. Dans la représentation des nombres signés, ce bit correspond au signe, d'oà le nom de flag Signe.

Dans le classement des instructions, nous vous présenterons dorénavant ainsi l'état d'un flag après une opération:

1 - Flag est mis après l'opération
0 - Flag est annulé après l'opération
I - Flag est inconnu après l'opération
X - Flag est mis ou annulé suivant le résultat de l'opération
P - Flag P/V indique la parité
 - (espace): aucune modification
! - Particularité

Exemple : Flags S Z P/V C
I x 1

signifie :

S - inconnu
Z - si 0, alors 1 et inversement
P/V - 1
C - aucune modification

Traduction Basic des instructions :

ADD A,H Basic: A=A+H

ADC A,&A9 Basic: A=A+&A9+CF

CF représente le flag Carry dont la valeur est également ajoutée.

SUB A,(HL) Basic: A=A-PEEK(HL)

SBC A,L Basic: A=A-L-CF

Exemples :

ADD A,(HL) A=&1F
HL=&B1C9

Case mémoire &B1C9: &43

```
  &1F = 0 0 0 1 1 1 1 1
+ &43 = 0 1 0 0 0 0 1 1
      ----------------------
          0 1 1 0 0 0 1 0
        8 7 6 5 4 3 2 1 0  - numéro de bit
```

Bit 8= 0 => flag Carry =0
Bit 7= 0 => flag Signe =0
Résultat <>0 => flag Zéro =0
Retenue externe =0 et retenue interne =0 => dépassement flag P/V =0

Contenu de l'accumulateur après l'opération : &X01100010 = &62

ADD A,D A contient &E1
D contient &A2

```
  &E1 = 1 1 1 0 0 0 0 1
+ &A2 = 1 0 1 0 0 0 1 0
  ---   --------------------
 &183 1 1 0 0 0 0 0 1 1
      8 7 6 5 4 3 2 1 0  - numéro de bit
```

Bit 8= 1 => flag Carry =1
Bit 7= 1 => flag Signe =1
Résultat <>0 => flag Zéro =0
Retenue externe et retenue interne => dépassement flag P/V =0

Contenu de l'accumulateur après l'opération : &X10000011 = &83

Comme vous le voyez, l'accumulateur ne contient pas le résultat correct. Il faut prendre en compte le flag Carry comme huitième bit pour obtenir le résultat correct. C'est pourquoi il est important d'examiner l'état des flags après des opérations arithmétiques pour corriger des résultats éventuellement incorrects.

Notez également que pour une addition dont le résultat est exactement 256, le flag Zéro est mis alors que le résultat n'est pas nul.

```
ADC A,&19 A=&5A
          Flag Carry= 1 (mis)

     &5A = 0 1 0 1 1 0 1 0
   + &19 = 0 0 0 1 1 0 0 1
           ---------------
     &74 = 0 1 1 1 0 1 0 0

   Flags :  S Z V C                Accu = &X01110100= &74
            0 0 0 0
```

Remarque : Si le Carry est annulé avant une instruction ADC, celle-ci est alors identique à ADD.

```
SUB A,(HL)  A contient &3C
            HL contient &BC19
            &BC19 contient &15

     0 0 1 1 0 1 1 0    &36
   + 1 1 1 0 1 0 1 1    complément à deux de &15
   -----------------
   1 0 0 1 0 0 0 0 1

Bit 7= 0 => flag Signe =0
Bit 8= 1 => flag Carry =0
```

Notez que c'est ici le complément de la retenue réelle qui a été pris (cas particulier !)

Résultat <>0 => flag Zéro =0

Retenue externe et retenue interne => dépassement flag P/V =0

Contenu de l'accumulateur après l'opération : &X00100001 = &21

```
SBC A,B A=&57
        B=&73
        flag Carry= 1 (mis)

      0 1 0 1 0 1 1 1 = &57
    + 1 0 0 0 1 1 0 1 = complément à deux de &73
    + 1 1 1 1 1 1 1 1 = complément à deux de &1 (CF)
    -----------------------
    1 1 1 1 0 0 0 1 1

    Flags :   S Z V C
              1 0 0 1

Accu = &X11100100= &E4
est le complément à deux de 29
le résultat est donc -29 (87-115-1=-29).
```

A côté de l'arithmétique binaire, il existe encore une autre possibilité de traiter des nombres dans un ordinateur :

Chaque chiffre du système décimal peut en effet être représenté par un bloc de 4 bits. Cette application est importante pour traiter les problèmes de gestion, pour lesquels il importe de respecter un nombre de chiffres et une précision donnés. Pour ce type d'opérations (BCD), il existe une instruction spéciale DAA qui prépare le contenu de l'accumulateur pour ces opérations.

Il y a en outre deux instructions spéciales que nous avons déjà évoquées, CPL et NEG.

CPL forme le complément du contenu de l'accumulateur et NEG le complément à deux du contenu de l'accumulateur.

Certaines instructions "normales" peuvent être également dénaturées en instructions spéciales. On peut par exemple utiliser SUB A pour vider le contenu de l'accumulateur. Cette instruction est en effet presque deux fois plus rapide et deux fois moins longue que LD A,0.

A ces instructions s'ajoutent encore les instructions de comptage qui augmentent ou diminuent la valeur d'une case mémoire ou d'un registre. Les instructions de comptage peuvent être employées avec les modes d'adressage implicite, de registre et indexé. Les instructions de ce type sont souvent utilisées pour la programmation des boucles. Leur mode de travail est simple :

```
INC x incrémente x et
DEC x décrémente x, x pouvant être :

reg, (HL), (XY+dis)

INC reg   Basic: reg=reg+1
DEC (HL)  Basic: POKE HL,PEEK(HL)-1
```

Les flags Signe, Zéro et V sont mis ou annulés en fonction du résultat de l'opération. Le carry n'est pas modifié. Il est important de noter que seules les instructions de comptage sur 8 bits modifient les flags. Pour les instructions de comptage sur 16 bits, il faut procéder en plus à une comparaison.

Liste d'instructions

ADD A,reg

ajouter le contenu du registre à celui de l'accumulateur et charger le résultat dans l'accumulateur.

A=A+reg

Code d'instruction :	10000rrr		Octet 1	Code d'opération

Flag : S Z V C
x x x x

ADD A,data

ajouter la constante au contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A+data

Code d'instruction :	11000110	&C6	Octet 1	Code d'opération
	<--co-->		Octet 2	Constante

Flag : S Z V C
x x x x

ADD A,(HL)

ajouter un octet de la mémoire au contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A+(HL)

Code d'instruction :	10000110	&86	Octet 1	Code d'opération

Flag : S Z V C
x x x x

ADD A,(XY+dis)

ajouter une case mémoire indexée au contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A+(XY+dis)

Code d'instruction :	11x11101		Octet 1	Code d'opération
	10000110	&86	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance

Flag : S Z V C
x x x x

ADC A,reg

ajouter le contenu du registre plus le bit Carry à celui de l'accumulateur et charger le résultat dans l'accumulateur.

A=A+reg+CF

Code d'instruction :	10001rrr	Octet 1	Code d'opération

Flag : S Z V C
x x x x

ADC A,data

ajouter la constante et le bit Carry au contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A+data+CF

Code d'instruction :	11001110	&CE	Octet 1	Code d'opération
	<--co-->		Octet 2	Constante

Flag :	S	Z	V	C
	x	x	x	x

ADC A,(HL)

ajouter un octet de la mémoire plus le bit Carry au contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A+prs+CF

Code d'instruction :	10001110	&8E	Octet 1	Code d'opération

Flag :	S	Z	V	C
	x	x	x	x

ADC A,(XY+dis)

ajouter une case mémoire indexée plus le bit Carry au contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A+CF+(XY+dis)

Code d'instruction :	11x11101		Octet 1	Code d'opération
	10001110	&8E	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance

Flag :	S	Z	V	C
	x	x	x	x

SUB reg

soustraire le contenu du registre de celui de l'accumulateur et charger le résultat dans l'accumulateur.

A=A-reg

Code d'instruction : 10010rrr Octet 1 Code d'opération

Flag : S Z V C
x x x x

SUB data

soustraire la constante du contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A-data

Code d'instruction : 11010110 &D6 Octet 1 Code d'opération
<--co--> Octet 2 Constante

Flag : S Z V C
x x x x

SUB (HL)

soustraire un octet de la mémoire du contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A-(HL)

Code d'instruction : 10010110 &96 Octet 1 Code d'opération

Flag : S Z V C
x x x x

SUB (XY+dis)

soustraire une case mémoire indexée du contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A-(XY+dis)

Code d'instruction :	11x11101		Octet 1	Code d'opération
	10010110	&96	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance

Flag : S Z V C
x x x x

SBC A,reg

soustraire le contenu du registre plus le bit Carry de celui de l'accumulateur et charger le résultat dans l'accumulateur.

A=A-reg-CF

Code d'instruction :	10011rrr	Octet 1	Code d'opération

Flag : S Z V C
x x x x

SBC A,data

soustraire la constante et le bit Carry du contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A-data-CF

Code d'instruction :	11011110	&DE	Octet 1	Code d'opération
	<--co-->		Octet 2	Constante

Flag : S Z V C
x x x x

SBC A,(HL)

soustraire un octet de la mémoire plus le bit Carry du contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A-prs-CF

Code d'instruction :	10001110	&8E	Octet 1	Code d'opération

Flag :	S	Z	V	C
	x	x	x	x

SBC A,(XY+dis)

soustraire une case mémoire indexée plus le bit Carry du contenu de l'accumulateur et charger le résultat dans l'accumulateur.

A=A-CF-(XY+dis)

Code d'instruction :	11x11101		Octet 1	Code d'opération
	10011110	&9E	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance

Flag :	S	Z	V	C
	x	x	x	x

DAA

convertir le contenu de l'accumulateur en format BCD.

Code d'instruction :	00100111	&27	Octet 1	Code d'opération

Flag :	S	Z	V	C	! : ces flags sont modifiés de façon différente
	!	x	!	!	par l'instruction spéciale DAA

CPL

former le complément de l'accumulateur.

(A=Pas A ou:) NOT A

Code d'instruction : 00101111 &2F Octet 1 Code d'opération

Flag : S Z V C

NEG

former la valeur négative (complément à deux) de l'accumulateur.

A=0-A (complément à deux de A)

Code d'instruction : 11101101 &ED Octet 1 Code d'opération
01000100 &44 Octet 2 Code d'opération

Flag : S Z V C
x x x ! ! : C est mis lorsque le contenu de l'accu <>0

INC reg

incrémenter le contenu du registre et charger le résultat dans le registre.

reg=reg+1

Code d'instruction : 00rrr100 Octet 1 Code d'opération

Flag : S Z V C
x x x

INC (HL)

incrémenter un octet de la mémoire et charger le résultat dans cet octet.

(HL)=(HL)+1

Code d'instruction :	00110100	&34	Octet 1	Code d'opération

Flag : S Z V C
x x x

INC (XY+dis)

incrémenter une case mémoire indexée et charger le résultat dans cette case mémoire.

(XY+dis)=(XY+dis)+1

Code d'instruction :	11x11101		Octet 1	Code d'opération
	10110100	&34	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance

Flag : S Z V C
x x x

DEC reg

décrémenter le contenu du registre et charger le résultat dans le registre.

reg=reg-1

Code d'instruction :	00rrr101	Octet 1	Code d'opération

Flag : S Z V C
x x x

DEC (HL)

incrémenter un octet de la mémoire et charger le résultat dans cet octet.

(HL)=(HL)-1

Code d'instruction :	00110101	&35	Octet 1	Code d'opération

Flag : S Z V C
x x x

DEC (XY+dis)

incrémenter une case mémoire indexée et charger le résultat dans cette case mémoire.

(XY+dis)=(XY+dis)-1

Code d'instruction :	11x11101		Octet 1	Code d'opération
	00110101	&35	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance

Flag : S Z V C
x x x

Instructions arithmétiques et de comptage sur 16 bits

Les instructions arithmétiques sur 16 bits restent pour le principe semblables aux instructions arithmétiques sur 8 bits. Les instructions sur 16 bits sont cependant plus limitées. Seules les instructions ADD, ADC et SUB existent pour certaines paires de registres. Le résultat d'une opération est toujours placé dans le registre HL (et non dans l'accumulateur comme pour les instructions sur 8 bits). Pour l'instruction ADD, il est également possible de stocker les résultats dans les registres d'index.

Les instructions 16 bits équivalent à plusieurs exécutions successives d'instructions 8 bits. Comme elles relient ces instructions automatiquement, elles sont plus rapides et plus courtes.

```
16 bits        8 bits

ADD HL,BC      LD  A,L
               ADD A,C
               LD  L,A
               LD  A,H
               ADC A,B
               LD  H,A
```

Toutes les instructions arithmétiques 16 bits utilisent l'adressage implicite. La modification des flags pour ADC et SBC est semblable à ce qu'elle est pour les instructions 8 bits. Pour ADD, seul le Carry est modifié et pour les instructions 16 bits INC et DEC, les flags ne sont pas du tout modifiés.

```
ADD IX,DE      Basic: IX=IX+DE
ADC HL,BC      Basic: HL=HL+BC+CF
SBC HL,SP      Basic: HL=HL-SP-CF
```

Exemple :

```
HL=&C000
DE=&0800


ADD HL,DE

  &C000  = 1100 0000 0000 0000
+ &0800  = 0000 1000 0000 0000
------------------------------
  &C800  = 1100 1000 0000 0000

Flag :  S Z V C
              0
```

Les flags S, Z et V ne sont pas modifiés

ADC HL,DE

```
 &F800   =   1111 1000 0000  0000
+ &0800  =   0000 1000 0000  0000
-----------------------------------------
 &10000  = 1 0000 0000 0000  0000

Flag :  S Z V C
        0 0 1 1
```

Ici aussi, HL ne reçoit pas le résultat correct &10000, mais 0. Le flag Carry permet de déceler cette erreur. Pour les opérations sur 16 bits, il représente le bit numéro 16 (=le dix-septième bit).

Les instructions de comptage 16 bits travaillent toutes avec l'adressage implicite. Elles peuvent se rapporter aux registres 16 bits BC, DE, HL, SP, IX et IY. Au contraire des instructions de comptage sur 8 bits, ces instructions ne modifient pas (!) les flags.

Liste d'instructions

ADD HL,prs

addition d'une paire de registres à HL.

HL=HL+ prs

Code d'instruction : 01pp1001 Octet 1 Code d'opération

```
Flag :  S Z V C
              x
```

ADC HL,prs

addition d'une paire de registres avec Carry à HL.

HL=HL+prs+CF

Code d'instruction :	11101101	&ED	Octet 1	Code d'opération
	01pp1010		Octet 2	Code d'opération

Flag : S Z V C
x x x x

SBC HL,prs

soustraction d'une paire de registres avec Carry de HL.

HL=HL-prs-CF

Code d'instruction :	11101101	&ED	Octet 1	Code d'opération
	01pp0010		Octet 2	Code d'opération

Flag : S Z V C
x x x x

ADD XY,prs

addition d'une paire de registres à XY.

XY=XY+prs

Code d'instruction :	11011101	&DD	Octet 1	Code d'opération
	00pp1001		Octet 2	Code d'opération

Flag : S Z V C
x

INC prs

incrémentation d'une paire de registres.

prs=prs+1

Code d'instruction :	00pp0011		Octet 1	Code d'opération

Flag : S Z V C

INC XY

incrémentation du registre d'index.

XY=XY+1

Code d'instruction :	11x11101		Octet 1	Code d'opération
	00100011	&23	Octet 2	Code d'opération

Flag : S Z V C

DEC prs

décrémentation d'une paire de registres.

prs=prs-1

Code d'instruction :	00pp1011		Octet 1	Code d'opération

Flag : S Z V C

DEC XY

décrémentation du registre d'index.

XY=XY+1

Code d'instruction :	11x11101		Octet 1	Code d'opération
	00101011	&2B	Octet 2	Code d'opération

Flag : S Z V C

Exercice :

Il est temps maintenant que nous utilisions enfin les nouvelles instructions que nous venons de découvrir. Ecrivez par exemple un petit programme d'addition de deux nombres 8 bits. Les nombres seront placés dans la Ram en Basic, par des instructions POKE. Le résultat de l'addition doit être stocké également dans la Ram. Après le retour au Basic, il pourra être lu et sorti grâce à une instruction PEEK.

Solution :

Comme les additions 8 bits utilisent systématiquement l'accumulateur, le premier opérande doit être chargé dans l'accumulateur :

```
LD A,opérande
```

Le second opérande sera stocké dans l'un des registres 8 bits :

```
LD H,opérande
```

Et nous pouvons maintenant opérer l'addition :

```
ADD A,H
```

Le résultat doit être placé dans la case mémoire &A100 :

```
LD (&A100),A
```

Si nous choisissons &A000 comme adresse de début, nous obtenons l'image suivante :

```
A000   3E10     10   LD   A,&10
A002   2620     20   LD   H,&20
A004   84       30   ADD  A,H
A005   3200A1   40   LD   (&A100),A
A008   C9       50   RET
```

La ligne DATA du programme Basic de chargement sera donc :

```
60 DATA &3E,&10,&26,&20,&84,&32,&00,&A1,&C9
```

Vous pouvez déduire du listing assembleur que le premier opérande se trouve à l'adresse &A001 et le second à l'adresse &A003. Dans le cas présent nous avons placé à ces adresses les valeurs &10 et &20. Le programme Basic qui fixe ces valeurs, exécute le programme en langage-machine et sort le résultat se présentera ainsi:

```
10 POKE &A001,operande1
20 POKE &A003,operande2
30 CALL &A000
40 PRINT PEEK (8A100)
```

4.7. INSTRUCTIONS LOGIQUES

Les instructions logiques sont également des instructions de traitement de données.

Le Z80 dispose des instructions logiques AND, OR et XOR (eXclusive OR) ainsi que de l'instruction de comparaison CP. Toutes ces instructions travaillent avec des données 8 bits. L'accumulateur est toujours le registre avec lequel sont effectuées les opérations logiques. C'est pourquoi l'accumulateur n'est pas indiqué comme opérande de l'instruction assembleur (AND B, alors que ADD A,B).

Les quatre instructions AND, OR, XOR et CP admettent les modes d'adressage suivants :

- implicite (registres A, B, C, D, E, H, L)
- indirect : registre (HL)
- indexé
- immédiat

Examinons les fonctions des instructions logiques. Chacun comprend l'affirmation logique suivante :

"S'il pleut, la rue se mouille."

Cette affirmation est un raisonnement de la forme <si ..., alors ...>. Voyons l'affirmation suivante :

"S'il pleut ET si je suis dans la rue, je me mouille."

Nous avons ici deux conditions reliées par ET. Le ET logique indique que les deux conditions, "il pleut" (première condition) et "je suis dans la rue" (deuxième condition) doivent être remplies pour que le résultat se produise. S'il ne pleut pas (première condition non remplie), je ne me mouille pas; si je suis dans une maison (deuxième condition non remplie), je ne me mouille pas non plus. Pour que la conséquence soit vraie, les deux conditions doivent donc être remplies. C'est exactement ce qui caractérise l'opération logique AND (ET). Comme l'ordinateur ne travaille qu'avec 0 et 1, on adopte la convention suivante :

1 signifie condition vraie
0 signifie condition fausse

Nous obtenons ainsi :

1 AND 1= 1	les deux conditions sont vraies	=> résultat vrai
1 AND 0= 0	une condition est fausse	=> résultat faux
0 AND 1= 0	une condition est fausse	=> résultat faux
0 AND 0= 0	les deux conditions sont fausses	=> résultat faux

Le Basic dispose également d'instructions logiques. Essayez-les :

```
PRINT 1 AND 1
PRINT 1 AND 0 etc...
```

Les opérations logiques sont très importantes en informatique. Elles sont en effet faciles à réaliser en électronique. On relie pour cela deux canaux d'entrée qui sont chargés de courant (=1) ou qui n'en sont pas chargés (=0) à un cercle de commutation dont le canal de sortie sera chargé ou non de courant (1 ou 0) en fonction des conditions en entrée. Un microprocesseur se compose d'un grand nombre de portes logiques connectées les unes à la suite des autres. L'addition dans la MPU est par exemple structurée en plusieurs opérations logiques.

Mais le programmeur n'entre jamais en contact avec ces structures. Nous appliquons les opérations logiques à des données (8 bits). Dans ce cas, ce sont toujours les bits de même rang des deux octets qui subissent l'opération logique :

```
        11111000
AND     01010011
        ---------
        01010000
```

```
Bit 0   : 0 AND 1=0
Bit 1   : 0 AND 1=0
Bit 2   : 0 AND 0=0
Bit 3   : 1 AND 0=0
Bit 4   : 1 AND 1=1
Bit 5   : 1 AND 0=0
Bit 6   : 1 AND 1=1
Bit 7   : 1 AND 0=0
```

Une des applications principales de l'instruction AND est la suppression ou le masquage de certains bits.

```
A=&X10111001
```

Supposons que nous ne voulions prendre en compte que les bits 0 à 3, donc que nous voulions masquer les bits 4 à 7. Pour cela, il nous faut "ANDer" (faire subir un ET logique) A avec &X00001111.

```
        10111001  :A
AND     00001111  :masque
        ---------
        00001001
```

Le masque que nous utilisons contient un 0 pour un bit à masquer et un 1 pour un bit significatif.

En Basic, nous aurions :

```
A=&X10111001
A=A AND &X00001111
```

Ce que nous traduirons ainsi en langage-machine :

```
LD A,&X10111001
AND  &X00001111
```

Considérez maintenant l'affirmation suivante :

"S'il pleut OU si je me baigne, je me mouille."

Le résultat est vrai lorsqu'au moins une des conditions est vraie. C'est le OU (OR) inclusif :

```
0 OR 0= 0
0 OR 1= 1
1 OR 0= 1
1 OR 1= 1
```

L'opération logique OR permet de mettre à 1 des bits déterminés d'un octet.

A contient &X10001011.

Supposons que nous voulions mettre à 1 les 3 bits supérieurs (5, 6 et 7) :

```
        10001011  :A
OR      11100000  :masque
        ---------
        11101011
```

Le masque contient un 1 pour chaque bit qui doit absolument être mis à 1 et un 0 pour les bits qui ne doivent pas être modifiés.

```
LD A,&X10001011       Basic: A=&X10001011
OR &X11100000         Basic: A=A OR &X11100000
```

Le XOR ou OU exclusif se distingue du ou inclusif par le fait que le résultat n'est positif que si une et seulement une des conditions est remplie. Si les deux conditions sont remplies, on aura 0. Le OU exclusif donne donc 1 pour des bits d'entrée différents et 0 pour des bits d'entrée identiques.

```
0 XOR 0= 0
1 XOR 0= 1
0 XOR 1= 1
1 XOR 1= 0
```

XOR a deux applications, la comparaison et la formation du complément. Les octets à comparer sont reliés avec XOR. Si le résultat est 0, c'est que les deux octets étaient identiques. S'il n'y avait pas identité, les bits différents sont mis.

```
        10101010
XOR     10101010  Comparaison ! !
        ---------
        00000000

        10101010
XOR     10101100  Comparaison ! !
        ---------
        00000110
```

=> les bits 1 et 2 des deux octets ne sont pas identiques.

Pour former le complément, on relie encore à un masque qui contient 1 pour un bit dont il faut former le complément et 0 pour 1 bit qui doit rester inchangé.

Il s'agit de former le complément des bits 4 à 7.

```
        10101111  :A
XOR     11110000  :masque
        ---------
        01011111
```

Comparaison avec le Basic :

Langage-machine	Basic
AND H	A=A AND H
OR (HL)	A=A OR PEEK(HL)
XOR &FF	A=A XOR &FF

Avec les instructions logiques, le carry est toujours annulé. Les flags Z et S sont modifiés comme d'habitude. Le flag P/V indique pour ces instructions la parité du résultat. La parité est 1 lorsque le nombre de 1 dans l'octet est pair et 0 lorsque ce nombre est impair.

Exercices :

1. Quel est l'effet de :

- OR avec &FF ?
- OR avec &0 ?
- AND avec &FF ?
- AND avec &0 ?
- XOR avec &FF ?
- XOR avec &0 ?

2. Le Basic dispose de l'instruction NOT. Traduisez cette instruction de deux façons différentes en langage-machine (par rapport à l'accumulateur).

Solution :

1.

```
OR &FF  => &FF, c'est-à-dire que tous les bits sont mis
OR &0   => aucune modification
AND &FF=> aucune modification
AND &0 => &0, c'est-à-dire que tous les bits sont annulés
XOR &FF=> formation du complément de tous les bits
XOR &0 => aucune modification
```

2.

```
Instruction XOR: XOR &FF
Instruction CPL: CPL
```

L'instruction de comparaison CP

L'instruction CP sert à comparer le contenu de l'accumulateur avec un octet. Cet octet peut être addressé de la façon suivante :

- implicite : registres A, B, C, D, E, H, L
- indirecte : paire de registres (HL)
- indexée
- immédiate

L'instruction CP ôte l'octet adressé de l'accumulateur et les flags sont modifiés en fonction du résultat du calcul. Au contraire de ce qui est le cas avec l'instruction SUB, le résultat n'est cependant pas sauvegardé dans l'accumulateur, le contenu de l'accumulateur n'est donc pas modifié par l'instruction. En fonction de l'état des flags, il est possible de faire exécuter un saut conditionnel à la suite de cette instruction.

Examinons les cas possibles avec une comparaison :

Le contenu de l'accumulateur est plus grand :

- dans ce cas, le carry est toujours à 0 car le résultat ne peut être supérieur à 255.

Le contenu de l'accumulateur est égal :

- dans ce cas, Z=1 puisque le résultat de la soustraction est 0. Dans ce cas également, le carry vaut 0 car il n'y a pas de retenue.

Le contenu de l'accumulateur est plus petit :

- dans ce cas, le flag carry est toujours mis puisqu'une retenue négative se produit.

Règles :

C=0 signifie >=
Z=0 signifie =
C=1 signifie <

nous obtenons ainsi :

Z=1 signigie <>
C=0 et Z=1 signifie >
C=1 ou Z=0 signifie =<

Ces règles ne valent que lorsque les octets à comparer sont considérés comme des nombres non signés entre 0 et 255.

Si les deux octets représentent des nombres signés en représentation en complément à deux, ce sont des règles plus complexes qui s'appliquent, règles qui découlent des règles pour les flags en arithmétique signée. Dans la plupart des cas, il n'est pas nécessaire d'avoir recours à ces règles.

Pour décider s'il y a égalité, on utilise le flag Z. La supériorité ou l'infériorité sont déterminées d'après l'état des flags S et V. Les flags S et V sont reliés entre eux avec XOR, c'est-à-dire que si V est mis (donc si une retenue s'est produite), S est inversé, sinon S reste inchangé.

```
S XOR V =0  signifie  >=
S XOR V =1  signifie  <
```

Nous supposerons pour la suite que les octets doivent être interprétés comme des nombres non signés.

Exemple :

```
A =&35
B =&21

CP B

donne   S Z V C
        0 0 0 0          parce que :

  00110101    : A
- 00100001    : B        (pas (!) complément à deux)
  ----------
  00010100

Pas de retenue           => C=0
Bit=0                    => S=0
  <>0                    => Z=0
pas de dépassement       => V=0
```

Le flag carry est nul. On en déduit que le contenu de l'accumulateur est plus grand que l'octet comparé (contenu du registre B).

```
C =&81

CP C  donne
```

```
Flag :  S Z V C
        1 0 1 1          parce que :

   00000001     : registre A
 - 10000001     : registre C
   ----------
  110000000
```

Retenue de 7 vers 8	=> C=1
Bit 7=1	=> S=1
<>	=> Z=0
Retenue de 7 vers 8 et pas de retenue de 6 vers 7	=> V=1

C=1, donc la valeur (contenu du registre C) avec laquelle s'est effectuée la comparaison était donc plus grande que le contenu de l'accumulateur.

Nous aurons par la suite à utiliser fréquemment l'instruction CP en liaison avec les instructions de tests et de sauts.

Liste d'instructions

AND reg

ANDer l'accumulateur avec un registre

A=A and reg

Code d'instruction : 10100rrr Octet 1 Code d'opération

```
Flag :  S Z P C
        x x x 0
```

AND data

ANDer l'accumulateur avec une constante

A=A and data

Code d'instruction :	11100110	&E6	Octet 1 Code d'opération
	<--co-->		Octet 2 Constante

Flag : S Z P C
x x x 0

AND (HL)

ANDer l'accumulateur avec une case mémoire

A=A and (HL)

Code d'instruction :	10100110	&A6	Octet 1 Code d'opération

Flag : S Z P C
x x x 0

AND (XY+dis)

ANDer l'accumulateur avec case mémoire indexée

A=A and (XY+dis)

Code d'instruction :	11x11101		Octet 1 Code d'opération
	10100110	&A6	Octet 2 Code d'opération
	<-dis-->		Octet 3 Distance

Flag : S Z P C
x x x 0

OR reg

ORer l'accumulateur avec un registre

A=A or reg

Code d'instruction :	10110rrr		Octet 1 Code d'opération

Flag : S Z P C
x x x 0

OR data

ORer l'accumulateur avec une constante

A=A or data

Code d'instruction :	11110110	&F6	Octet 1 Code d'opération
	<--co-->		Octet 2 Constante

Flag : S Z P C
x x x 0

OR (HL)

ORer l'accumulateur avec une case mémoire

A=A or (HL)

Code d'instruction :	10110110	&B6	Octet 1 Code d'opération

Flag : S Z P C
x x x 0

OR (XY+dis)

ORer l'accumulateur avec case mémoire indexée

A=A or (XY+dis)

Code d'instruction :	11x11101		Octet 1 Code d'opération
	10110110	&B6	Octet 2 Code d'opération
	<-dis-->		Octet 3 Distance

Flag : S Z P C
x x x 0

XOR reg

Effectuer un OU exclusif entre l'accumulateur et un registre

A=A xor reg

Code d'instruction :	10101rrr		Octet 1 Code d'opération

Flag : S Z P C
x x x 0

XOR data

Effectuer un OU exclusif entre l'accumulateur et une constante

A=A xor data

Code d'instruction :	11101110	&EE	Octet 1 Code d'opération
	<--co-->		Octet 2 Constante

Flag : S Z P C
x x x 0

XOR (HL)

Effectuer un OU exclusif entre l'accumulateur et une case mémoire

A=A xor (HL)

Code d'instruction :	10101110	&AE	Octet 1 Code d'opération

Flag : S Z P C
x x x 0

XOR (XY+dis)

Effectuer un OU exclusif entre l'accumulateur et une case mémoire indexée

A=A xor (XY+dis)

Code d'instruction :	11x11101		Octet 1 Code d'opération
	10101110	&AE	Octet 2 Code d'opération
	<-dis-->		Octet 3 Distance

Flag : S Z P C
x x x 0

CP reg

Comparer l'accumulateur avec un registre

A-reg

Code d'instruction :	10111rrr		Octet 1 Code d'opération

Flag : S Z P C
x x x x

CP data

Comparer l'accumulateur avec une constante

A-data

Code d'instruction :	11111110	&FE	Octet 1 Code d'opération
	<--co-->		Octet 2 Constante

Flag : S Z P C
x x x x

CP (HL)

Comparer l'accumulateur avec une case mémoire

A-(HL)

Code d'instruction :	10111110	&BE	Octet 1 Code d'opération

Flag : S Z P C
x x x x

CP (XY+dis)

Comparer l'accumulateur avec case mémoire indexée

A-(XY+dis)

Code d'instruction :	11x11101	&DD	Octet 1 Code d'opération
	10111110	&BE	Octet 2 Code d'opération
	<-dis-->		Octet 3 Distance

Flag : S Z P C
x x x x

Le programme de démonstration :

```
A000 06FF        10    LD   B,&FF
A002 2100C0      20    LD   HL,&C000
A005 7E          30    LD   A,(HL)
A006 A8          40    XOR  B
A007 77          50    LD   (HL),A
A008 23          60    INC  HL
A009 3E00        70    LD   A,0
A00B BC          80    CP   H
A00C 20F7        90    JR   NZ,&A005
A00E C9         100    RET
```

Ce programme inverse le contenu de tout l'écran, en mode 2.

LD B,&FF est le masque avec lequel l'instruction XOR B inversera les contenus successifs de l'accumulateur.

L'adresse de début de l'écran &C000 est chargée dans HL (LD HL,&C000). Puis commence la boucle du programme. LD A,(HL) lit un octet dans la mémoire écran. XOR B inverse cet octet et LD (HL),A le réécrit dans la mémoire écran. HL est alors incrémenté (INC HL)et on teste si HL est encore à l'intérieur de la mémoire écran.

HL parcourt les valeurs &C000 à &FFFF. Si HL est alors incrémenté encore une fois, (&FFFF+1), HL reçoit la valeur 0. Le résultat devrait normalement être &10000 mais comme HL ne peut stocker que des nombres 16 bits, le bit supplémentaire est négligé, donc HL=0.

L'instruction CP doit nous permettre de déterminer si HL vaut déjà 0. Comme CP compare toujours avec le contenu de l'accumulateur, il faut d'abord charger 0 dans l'accumulateur, avec LD A,0.

Pour cette comparaison, il suffit de comparer l'octet fort de HL. Dans le cas présent en effet, si H=0, c'est que HL=0. A la suite de l'instruction CP, le flag Z indique si HL est déjà nul ou non.

L'instruction de saut qui suit, JR NZ,&A005 signifie :

"Sauter à l'adresse &A005 si Z n'est pas nul (Non Zero), sinon exécuter la prochaine instruction."

Si HL=0, le programme se termine par RET.

Les lignes de DATA pour le programme de chargement Basic :

```
DATA &06,&FF,&21,&00,&C0,&7E,&A8,&77
DATA &23,&3E,&00,&BC,&20,&F7,&C9
```

Choisissez &A000 comme adresse de départ, &A000+15-1=&A00E comme adresse de fin et lancez le programme avec >CALL &A000< (en MODE 2).

Vous pouvez également remplacer l'instruction XOR B par CPL (former le complément de l'accumulateur).

Passez maintenant en MODE 1 et essayez la routine. Le résultat est inattendu. Cela vient de la structure de la mémoire écran. Comme vous le savez, en MODE 2, les bits mis correspondent directement aux points allumés sur l'écran. C'est pourquoi il n'est pas possible d'avoir plusieurs couleurs d'écriture en MODE 2. En MODE 1, vous disposez de quatre couleurs. Comme seule est disponible pour les informations sur l'écran la zone de &C000 à &FFFF et qu'il faut encore y stocker les informations sur les couleurs, en MODE 1, les quatre bits supérieurs de chaque octet servent chacun à fixer un double-point sur l'écran. Les bits inférieurs déterminent la couleur. Comme ce sont les points et non les couleurs que nous voulons inverser, il nous faut changer le masque d'inversion. LD B,&FF (&FF=&X11111111) signifie que tous les bits seront inversés par XOR B. Avec &X11110000=&F0, seuls les quatre bits supérieurs seront inversés.

Pour utiliser le programme également en MODE 1, il nous faut donc remplacer la seconde valeur de la ligne de DATA, &FF, par &F0. En MODE 0, seuls les bits 6 et 7 correspondent aux points de l'écran. Procédez vous-même à la modification nécessaire dans le programme.

4.8. INSTRUCTIONS DE ROTATION ET DE DECALAGE

Que signifie le décalage des chiffres d'un nombre ?

10^4	10^3	10^2	10^1	10^0	
	3	7	3	0	
3	7	3	0	0	: décalage vers la gauche !
		3	7	3	: décalage vers la droite !

En système décimal, un décalage vers la gauche se traduit par une multiplication par 10 (base du système décimal) et un décalage vers la droite par une division par 10. (Un décalage des chiffres vers la gauche correspond également à un décalage de la virgule vers la droite).

De même, un décalage en système binaire correspond à une division ou à une multiplication par 2 (la base du système). Il n'y a pas en Basic d'équivalent direct de ces instructions mais celles-ci peuvent être traduites par les instructions de multiplication et de division par 2.

Le Z80 dispose de 76 instructions de ce type dont la plupart utilisent les modes d'adressage implicite, indirect ou indexé. Il y a plusieurs sortes de rotations et de décalages. Mais voyons d'abord quelle est la différence entre les opérations de décalage et de rotation.

Décalage : le décalage consiste à déplacer bit par bit le contenu du registre concerné vers la droite ou vers la gauche. Le bit qui, à une extrêmité de l'octet, est "expulsé" de l'octet par ce mouvement est placé dans le carry. L'emplacement libéré à l'autre extrêmité de l'octet est rempli par un 0.

L'utilisation de l'instruction SRL avec des nombres signés débouche sur une erreur. Le bit 7 est en effet remplacé par un 0. Or cela aurait pour effet de transformer un nombre négatif (bit 7=1) en un

nombre positif (bit 7=0). L'instruction SRA permet d'éviter cette erreur. Avec cette instruction, le bit remplaçant le bit 7 est identique au bit signe, c'est-à-dire à l'ancien état du bit 7. Ce bit est donc 0 lorsque le bit le plus à gauche était auparavant égal à 0 (nombre positif) et 1 lorsque le bit de gauche était égal à 1 (signe -). Comme cette instruction prend en compte la signification arithmétique du bit 7, on le qualifie d'instruction arithmétique (et non logique) de décalage.

Rotation : Au contraire du décalage, la rotation remplace le bit à combler soit par le bit qui a été expulsé de l'autre côté, soit par le bit Carry.

Sur le Z80, il y a deux sortes de rotation :

Rotation sur 8 bits (sans Carry)
Rotation sur 9 bits (avec Carry)

Pour une rotation sur 9 bits, les 8 bits sont décalés d'un emplacement vers la droite. Le bit expulsé sur la droite est placé dans le Carry. Le bit entrant pour combler l'espace libéré sur la gauche est l'ancien contenu du Carry (avant qu'il n'ait été remplacé par le bit qui vient d'être expulsé). Il s'agit donc bien d'une rotation sur 9 bits puisqu'elle affecte les 8 bits de l'octet ainsi que le Carry (le neuvième bit!).

La rotation sur 8 bits n'affecte que les 8 bits du registre concerné. Le bit expulsé est bien placé dans le Carry, mais le Carry ne participe pas à la rotation. Le bit expulsé d'un côté va en effet combler l'espace libéré de l'autre côté.

Il y a encore deux instructions spéciales pour la rotation des chiffres (=blocs de 4 bits) en format BCD.

RLD et RRD (D:digit=chiffre) font subir une rotation à deux chiffres de la case mémoire indiquée par HL ainsi qu'au chiffre que constitue la moitié inférieure de l'accumulateur.

Les instructions de rotation et de décalage ont le plus souvent un code d'opération sur deux octets. Le premier octet du code d'opération est toujours &CB. (Pour les instructions avec adressage indexé, &CB est le second octet car le premier octet, avec ce mode d'adressage, est &DD ou &FD. Exception: RRD/RLD commencent par ED.) Comme les instructions de rotation sont souvent utilisées pour l'arithmétique, quatre instructions supplémentaires ont été créées qui ne se rapportent qu'à l'accumulateur et qui ont un code d'opération sur un octet. Ces instructions sont exactement deux fois plus courtes et deux fois plus rapides d'exécution que les instructions normales :

"Normal"	"Spécial-accu"
RLC A	RLCA
RRC A	RRCA
RL A	RLA
RR A	RRA

Les instructions normales de rotation ou de décalage modifient les flags S et Z comme à l'habitude. Le flag P/V indique la parité. Le Carry contient le bit expulsé. Les instructions spéciales pour l'accumulateur ne modifient pas S, Z et P/V. Les instructions de rotation en BCD, soit RLD et RRD modifient les flags S, Z et P comme les autres instructions mais pas le Carry.

Exemples :

```
SRL C       C:&36

      00110110                  :&36
   0--> 0011011 -->   0 dans le Carry
      00011011                  :registre C après exécution
                      0         :le Carry après exécution
```

SRL se traduit par une division par 2 : &36/2=&1B

```
    SRA (HL)  HL:&B100
              Case mémoire &B100:&C2

        11000010                  :&C2
        *1100001 -->    0 dans le Carry
        11000010        0         :CF. (HL) après exécution = &E1
                        0         :le Carry après exécution

    (* le bit 7 reste inchangé)
```

En complément à deux,

```
    &C2  signifie -62
    &E1  signifie -31
```

L'instruction SRA effectue correctement la division par deux des nombres signés. SRL (HL) aurait par contre eu pour résultat &61=97. Ce qui n'est pas la moitié de -62, mais celle de 194 ce qui correspond au nombre non signé &C2.

```
    RLC D     D:&E4
              Carry=1

        &E4 = &X11100100
nouveau Carry <-- 11100100 <--  1=ancien Carry
                  11001001
Contenu de D après exécution : &C9
Flag Carry (CF) = 1
```

&C9 n'est cependant pas le double de &E4. La raison en est qu'un bit a été décalé dans le Carry. C'est donc &1C9 qui est censé être le double de &E4. Cela n'est pas entièrement exact puisque l'ancien Carry (=1) a également participé à la rotation. C'est donc &1C9-1=&1C8 qui représente le double de &E4.

Pour faire subir une rotation à des nombres composés de plusieurs octets, on utilise RLC ou RRC qui fait participer le bit expulsé lors de la rotation du dernier octet à la rotation de l'octet suivant, à travers le Carry. (voir le programme à la fin du chapitre).

```
RRA (HL)  accu:&7600

  &76 = &X011101110          :&C2
        &X*01110110 -> dans le Carry
(ici entre l'ancien bit 0)
Accu : &X00111011      CF=0

Contenu de l'accumulateur :        &3B
                                   &3B*2=&76
```

Liste d'instructions

RLCA

Rotation vers la gauche de l'accumulateur (8 bits)

Code d'instruction :   00000111   &07   Octet 1   Code d'opération

Flag :   S Z V C
              x   Contenu du bit 7 venant de A

RLA

Rotation vers la gauche de l'accumulateur à travers le Carry (9 bits)

Code d'instruction :   00010111   &17   Octet 1   Code d'opération

Flag :   S Z V C
              x   Contenu du bit 7 venant de A

RRCA

Rotation vers la droite de l'accumulateur (8 bits)

Code d'instruction :	00001111	&0F	Octet 1	Code d'opération

Flag : S Z V C
x Contenu du bit 0 venant de A

RRA

Rotation vers la droite de l'accumulateur à travers le Carry (9 bits)

Code d'instruction :	00011111	&1F	Octet 1	Code d'opération

Flag : S Z V C
x Contenu du bit 0 venant de A

RLC reg

Rotation vers la gauche d'un registre (8 bits)

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00000rrr		Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 7

RLC (HL)

Rotation vers la gauche d'une case mémoire (8 bits)

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00000110	&06	Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 7

RLC (XY+dis)

Rotation vers la gauche d'une case mémoire indexée (8 bits)

Code d'instruction :	11x11101		Octet 1	Code d'opération
	11001011	&CB	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance
	00000110	&06	Octet 4	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 7

RL reg

Rotation vers la gauche d'un registre à travers le Carry (9 bits)

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00010rrr		Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 7

RL (HL)

Rotation vers la gauche d'une case mémoire à travers le Carry (9 bits)

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00010110	&16	Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 7

RL (XY+dis)

Rotation vers la gauche d'une case mémoire indexée, à travers le Carry (9 bits)

Code d'instruction :	11x11101		Octet 1	Code d'opération
	11001011	&CB	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance
	00010110	&16	Octet 4	Code d'opération

Flag :	S	Z	V	C	
	x	x	x	x	Contenu du bit 7

RRC reg

Rotation vers la droite d'un registre (8 bits)

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00001rrr		Octet 2	Code d'opération

Flag :	S	Z	V	C	
	x	x	x	x	Contenu du bit 0

RRC (HL)

Rotation vers la droite d'une case mémoire (8 bits)

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00001110	&0E	Octet 2	Code d'opération

Flag :	S	Z	V	C	
	x	x	x	x	Contenu du bit 0

RRC (XY+dis)

Rotation vers la droite d'une case mémoire indexée (8 bits)

Code d'instruction :	11x11101		Octet 1	Code d'opération
	11001011	&CB	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance
	00001110	&0E	Octet 4	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 0

RR reg

Rotation vers la droite d'un registre à travers le Carry (9 bits)

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00011rrr		Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 0

RR (HL)

Rotation vers la droite d'une case mémoire à travers le Carry (9 bits)

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00011110	&1E	Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 0

RR (XY+dis)

Rotation vers la gauche d'une case mémoire indexée, à travers le Carry (9 bits)

Code d'instruction :	11x11101		Octet 1	Code d'opération
	11001011	&CB	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance
	00011110	&1E	Octet 4	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 0

SLA reg

Décalage vers la gauche d'un registre

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00100rrr		Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 7

SLA (HL)

Décalage vers la gauche d'une case mémoire

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00100110	&26	Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 7

SLA (XY+dis)

Décalage vers la gauche d'une case mémoire indexée

Code d'instruction :				
	11x11101		Octet 1	Code d'opération
	11001011	&CB	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance
	00100110	&26	Octet 4	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 7

SRA reg

Décalage "arithmétique" vers la droite d'un registre

Code d'instruction :				
	11001011	&CB	Octet 1	Code d'opération
	00101rrr		Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 0

SRA (HL)

Décalage "arithmétique" vers la droite d'une case mémoire

Code d'instruction :				
	11001011	&CB	Octet 1	Code d'opération
	00101110	&2E	Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 0

SRA (XY+dis)

Décalage "arithmétique" vers la droite d'une case mémoire indexée

Code d'instruction :	11x11101		Octet 1	Code d'opération
	11001011	&CB	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance
	00101110	&2E	Octet 4	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 0

SRL reg

Décalage vers la droite d'un registre

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00111rrr		Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 0

SRL (HL)

Décalage vers la droite d'une case mémoire

Code d'instruction :	11001011	&CB	Octet 1	Code d'opération
	00111110	&3E	Octet 2	Code d'opération

Flag : S Z V C
x x x x Contenu du bit 0

SRL (XY+dis)

Décalage vers la droite d'une case mémoire indexée

Code d'instruction :	11x11101		Octet 1	Code d'opération
	11001011	&CB	Octet 2	Code d'opération
	<--dis->		Octet 3	Distance
	00111110	&2E	Octet 4	Code d'opération

Flag :	S	Z	V	C	
	x	x	x	x	Contenu du bit 0

RLD

Rotation sur 4 bits (nibble swap) vers la gauche entre accumulateur et mémoire

Code d'instruction :	11101101	&ED	Octet 1	Code d'opération
	01101111	&6F	Octet 2	Code d'opération

Flag :	S	Z	V	C
	x	x	x	

RRD

Rotation sur 4 bits (nibble swap) vers la droite entre accumulateur et mémoire

Code d'instruction :	11101101	&ED	Octet 1	Code d'opération
	01100111	&67	Octet 2	Code d'opération

Flag :	S	Z	V	C
	x	x	x	

Programmes

L'application la plus usuelle des instructions de rotation et de décalage est le calcul. Nous allons donc "dénaturer" quelque peu ces instructions et les utiliser pour décaler l'écran.

Avec l'aide des instructions de transfert de bloc nous avons réussi à décaler l'écran horizontalement, caractère par caractère. Les nouvelles instructions vont nous permettre de réaliser un décalage bit par bit.

Le listing assembleur :

```
A000 97          XOR A
A001 2100C0      LD HL,&C000
A004 CB3E        SRL (HL)
A006 23          INC HL
A007 BC          CP H
A008 20FA        JR NZ,&A004
A00A C9          RET
```

Vous reconnaissez là la structure de base de la boucle avec laquelle nous faisions prendre à HL les valeurs &C000 à &FFFF.

La première instruction est nouvelle.

XOR A remplace l'instruction normalement utilisée LD A,0. XOR A vide l'accumulateur. Cette instruction est plus rapide du fait de son mode d'adressage implicite.

LD A,0 est une instruction à adressage immédiat, c'est-à-dire que les données (0!) doivent être lues. Et voici maintenant le point le plus important du programme :

```
SRL (HL)
```

Comme HL doit parcourir la totalité de la zone d'adresses, nous avons choisi le mode d'adressage indirect. SRL décale les 8 bits de chaque octet de l'écran d'un emplacement vers la droite.

Convertissez ce programme en lignes de DATA, en vous aidant du listing assembleur et chargez-le avec le programme de chargement Basic à partir de l'adresse &A000. Le programme décale chaque caractère de l'écran vers la droite. Comme nous ne prenons pas en compte le bit expulsé sur la droite, les caractères perdent leur bit le plus à droite.

Entrez maintenant :

```
FOR I=1 TO 8:CALL &A000:NEXT
```

Cette instruction a pour effet de vider l'écran. Les caractères disparaissent en effet bit par bit vers la droite puisqu'avec l'instruction SRL, le bit rentrant par la gauche est 0 (=pas de point). Remplaçons SRL (HL) par SLA (HL).

Le code pour cette instruction est &CB,&25. Remplacez donc dans les lignes DATA le 5ème élément (&3E) par &25 et chargez à nouveau le programme avec RUN. Ce programme se traduit par un décalage semblable, mais vers la gauche.

Essayez également SRA (HL), code: &CB, &2E. Le 5ème octet en lignes de DATA sera alors &2E. Si vous entrez à nouveau :

```
FOR I=1 TO 8:CALL &A000:NEXT
```

vous obtenez une curieuse image à l'écran. C'est qu'en effet l'instruction SRA ne touche pas au bit 7. Après que vous ayez fait exécuté plusieurs fois cette instruction, tous les bits se trouvent avoir la valeur qu'avait le bit 7 auparavant.

La lettre R du message Ready se transforme (après exécution par 8 fois) en deux traits horizontaux. Cela vient de la carte bits de cette lettre.

```
               76543210    Numéro de bit
              1******
              2 **   **
              3 **   **
Ligne         4 *****
```

```
5 ** **
6 **  **
7***   *
8
```

Chaque caractère est représenté de la même façon dans une grille 8x8. Pour le R, le bit 7 n'est mis qu'en lignes 7 et 1. Si vous appelez maintenant par 8 fois consécutives votre programme en langage-machine, vous pouvez constater que le R disparaît et que ne subsistent que deux traits en lignes 1 et 7.

Le e de Ready disparaît totalement car pour ce caractère, aucun bit 7 n'est mis. Du a, il reste un trait en ligne 6, du d des traits en lignes 4, 5 et 6 et de y il ne reste rien.

Essayez de bien comprendre, au vu de ce résultat, pourquoi l'instruction SRA est qualifiée d'instruction arithmétique, alors que l'instruction SRL est qualifiée d'instruction logique. Essayez d'employer également les autres instructions dans le programme.

RRC a pour code &CB, &DE; RLC a pour code &CB,&06.

Modifiez le programme de chargement et faites exécuter le programme 8 fois avec la boucle FOR-NEXT. Vous pouvez ainsi voir clairement pourquoi ces instructions sont qualifiées d'instructions de rotation. Chaque caractère subit une rotation, c'est-à-dire que les bits qui sont expulsés sur la droite ou la gauche (pour RRC ou RLC) sont à nouveau introduits de l'autre côté. Après exécution par 8 fois, l'écran se trouve à nouveau dans sa position de départ.

Restent encore les instructions de la rotation sur 9 bits, RL (code &CB, &16) et RR (code &CB,&1E).

En appelant le programme avec RR, l'écran se remplit de rayures. Chaque fois que vous appelez le programme, ces rayures s'étendent, jusqu'à ce qu'après 8 appels l'écran soit entièrement blanc. Ce n'est cependant pas du tout le résultat attendu. Par la rotation sur 9 bits, le contenu de l'écran aurait dû être décalé dans l'une ou l'autre direction.

Le contenu devrait avoir été décalé d'un bit vers la droite puisque le bit expulsé est stocké dans le carry et rentre dans l'octet suivant lors de sa rotation.

Mais comme nous n'avons pas obtenu le résultat escompté, il y a manifestement une erreur dans notre programme.

Essayez de trouver cette erreur et réfléchissez à une solution possible !

(un conseil : examinez la modification des flags!)

Comme le premier bit d'un caractère est mis après chaque exécution de l'instruction (d'où les traits sur l'écran) et que ce bit vient du flag Carry, cela signifie que le Carry était toujours à 1. Il ne correspondait donc pas au dernier bit de l'octet précédent. Comment cela est-il possible ?

Considérons les autres instructions du programme. Après la rotation vient l'instruction INC. Les instructions de comptage sur 16 bits ne modifient pas les flags. Mais ensuite vient CP H et c'est là qu'est l'erreur !

La tâche de l'instruction CP consiste justement à modifier les flags. Chaque fois que la boucle est parcourue, cette instruction modifie le Carry. Comme H est plus grand que A (A=0), le carry est mis chaque fois, sauf lors du premier parcours. Le flag carry mis participe ensuite à la rotation sur l'écran effectuée par RR et l'écran devient blanc.

Pour résoudre ce problème, il y a deux solutions possibles :

1. - stockage provisoire des flags avant chaque instruction CP
2. - contourner la modification des flags

En ce qui concerne la première solution : les instructions de pile permettent de stocker le registre flags sur la pile (immédiatement après l'instruction de rotation) et d'aller ensuite le retirer de la pile (juste avant l'instruction de rotation). Il faut donc ajouter PUSH AF (=sauvegarder sur la pile) après RR et POP AF

(=retirer de la pile) avant RR. Il nous faut également veiller à ce que la pile fonctionne correctement.

La première instruction de pile dans notre programme serait, suivant ce que nous venons de dire, POP AF. Ce serait cependant tout à fait incorrect puisque cette instruction tenterait de lire la première fois des informations qui ne sont pas encore sur la pile. Au lieu de ces informations, c'est l'adresse de retour qui serait retirée de la pile. Lorsqu'il s'agirait ensuite de retourner au Basic, le programme sauterait donc à une adresse erronnée. C'est pourquoi il faut ajouter avant le début de la boucle une instruction PUSH AF ainsi qu'une instruction POP AF après la boucle (avant RET).

Lorsque vous utilisez les instructions PUSH et POP, faites très attention à ce qu'elles figurent toujours dans l'ordre qui convient: PUSH POP. Avec nos dernières améliorations, le programme se présente maintenant ainsi :

```
A000 97         10    SUB A,A
A001 F5         15    PUSH AF
A002 2100C0     20    LD HL,&C000
A005 F1         25    POP AF
A006 CB1E       30    RR (HL)
A008 F5         35    PUSH AF
A009 23         40    INC HL
A00A BC         50    CP H
A00B 20FB       60    JR NZ,&A005
A00D F1         65    POP AF
A00E C9         70    RET
```

Même si un programme Basic met une minute à exécuter la même tâche de décalage sur 1 bit, ce programme en langage-machine est presque déjà trop lent. Les deux instructions de pile dans la boucle qui est parcourue 16000 fois ralentissent le programme inutilement. Pour remédier à ce qui constitue un inconvénient au niveau de la vitesse d'exécution, nous vous proposons la deuxième solution.

En ce qui concerne la deuxième solution : pour que l'instruction JR NZ fonctionne mais que le flag Carry reste malgré tout inchangé, il nous faut utiliser une instruction qui affecte le flag Z mais pas le flag Carry. Les instructions de comptage sur 8 bits répondent justement à cette exigence. Pour incrémenter la paire de registres HL, il nous faut utiliser deux instructions de comptage sur 8 bits. Nous incrémentons tout d'abord l'octet faible. Si L est différent de 0 après exécution de cette instruction, la boucle est répétée.
Si L vaut 0, H doit être incrémenté de 1.

Exemple :

	H=&C0	L=&FE	HL=&C0FE
après incrémentation :	H=&C0	L=&FF	HL=&C0FF
après incrémentation :	H=&C1	L=&0	HL=&C100

La section de programme modifiée :

```
INC L
JR NZ,adresse
INC H
JR NZ,adresse
RET
```

L'instruction SUB A peut d'autre part être supprimée puisque l'accumulateur n'est plus utilisé.

Listing assembleur :

```
A000 2100C0     10      LD HL,&C000
A003 CB1E       20      RR (HL)
A005 2C         30      INC L
A006 20FB       40      JR NZ,&A003
A008 24         50      INC H
A009 20F8       60      JR NZ,&A003
A00A C9         70      RET
```

Convertissez le programme en lignes de DATA :

```
60 DATA &21,&00,&C0,&CB,&1E,&2C,&20,&FB
70 DATA &24,&20,&F8,&C9
```

Chargez ensuite le programme avec RUN et essayez-le.

L'instruction RRD: si vous remplacez &CB (octet 4) par &ED et &1E par &67, le programme exécutera un décalage de 4 bits (=un chiffre BCD).

Essayez le programme Basic suivant :

```
5 MODE 2
10 FOR K=1 TO 4
20 FOR I=0 TO 11
30 LOCATE (K-1)*8+1,12-I:PRINT"SALUT"
35 LOCATE (K-1)*8+1,12+I:PRINT"SALUT"
40 FOR J=1 TO K
50 CALL &A000
60 NEXT J
70 NEXT I
80 NEXT K
```

4.9. INSTRUCTIONS DE MANIPULATION DE BITS

Nous avons montré au chapitre 4.7 que les instructions logiques peuvent être utilisées pour sélectionner ou pour annuler des bits isolés ou des groupes de bits de l'accumulateur. Il est cependant intéressant de pouvoir utiliser une instruction qui permette de sélectionner ou d'annuler n'importe quel bit de n'importe quel registre ou case mémoire. Comme une telle instruction ne peut elle-même être construite qu'avec un nombre important d'instructions, la plupart des processeurs ne proposent que peu ou pas du tout d'instructions de manipulation de bits. Sur ce plan, le Z80 est richement doté. Si l'on inclut les instructions de test sur les bits, le Z80 dispose de 120 instructions de manipulation de bits.

Les instructions de test sur les bits examinent si un bit donné d'un registre ou d'une case mémoire est mis ou annulé. Suivant le résultat de ce test, le flag Zéro est mis ou annulé. Le Carry n'est pas modifié, l'état des flags S et P/V après exécution est indéterminé (!). Les deux instructions pour mettre (SET) et pour annuler (RES) un bit n'affectent nullement les flags.

Toutes les instructions bits commencent par le code d'opération &CB (comme toujours, excepté pour les instructions avec adressage indexé). Le second code d'opération est donné par le numéro de bit et le code du registre.

Pour l'adressage de l'octet concerné, on dispose des modes d'adressage suivants :

- implicite : registres A, B, C, D, E, H, L
- indirect : (HL)
- indexé : (XY+dis)

Format :

```
BIT b,reg      BIT b,(HL)      BIT b,(XY+dis)
RES b,reg      RES b,(HL)      RES b,(XY+dis)
SET b,reg      SET b,(HL)      SET b,(XY+dis)

b=numéro de bit
```

Le numéro de bit b est ainsi codé :

```
0 - 000      4 - 100
1 - 001      5 - 101
2 - 010      6 - 110
3 - 011      7 - 111
```

Ces instructions sont également qualifiées d'instructions avec adressage par bit puisque le bit à appeler est indiqué dans le code d'opération.

Exemples :

BIT 6,B B:&33

```
&X00110011  =&33
  *
  76543210  - numéro de bit
```

*: le bit numéro 6 vaut 0
Le flag Z est mis à 1 puisque le bit 6=0.
Après exécution :

```
B=&33    Flag : S Z V C
                I 1 I    I:flags S et V sont inconnus
```

RES 1,(HL) HL:&A975
(&A975)=&23

```
&X00100011  =&23
        *
  76543210  - numéro de bit
```

*: le bit numéro 1 est annulé

&X00100001 =&21

Case mémoire &A975 après exécution : &21

```
Flag : S Z V C
                 - aucune modification
```

SET 7,C C:&7F

```
&X01111111  =&7F
  *
  76543210  - numéro de bit
```

*: le bit numéro 7 est mit

```
&X11111111  =&FF
```

Registre C après exécution : &FF.

Flag: S Z V C
- aucune modification

Comparaison avec le Basic

Essayons de reproduire l'instruction SET b,a en Basic: le bit b doit être mis. L'instruction OR nous permet de mettre des bits déterminés. Le b-ième bit vaut 2^b. Donc :

```
SET b,A      Basic: A=A OR (2^b)
```

De même, pour RES :

```
RES b,A      Basic: A=A AND (255-2^b)
```

Les instructions spéciales SCF et CCF :

Comme le bit 0 du registre F (le Carry) est utilisé particulièrement souvent, deux instructions spéciales le concernent :

SCF met le Carry à 1.

CCF inverse la valeur du flag Carry: C=0 => C=1 et C=1 => C=0.

Ce sont les seules instructions qui permettent une manipulation directe des flags.

Les instructions logiques permettent d'annuler le Carry.

Liste d'instructions

Dans la liste d'instructions, b représente le numéro du bit concerné. Dans le code d'opération, le code pour le numéro de bit est figuré par 'bbb'.

BIT b,reg

Tester un bit d'un registre

Code d'instruction :	11001011	&CB	Octet 1 Code d'opération
	01bbbrrr		Octet 2 Code d'opération

Flag : S Z V C
I x I

BIT b,(HL)

Tester un bit d'une case mémoire

Code d'instruction :	11001011	&CB	Octet 1 Code d'opération
	01bbb110		Octet 2 Code d'opération

Flag : S Z V C
I x I

BIT b,(XY+dis)

Tester un bit d'une case mémoire avec adressage indexé

Code d'instruction :	11x11101		Octet 1 Code d'opération
	11001011	&CB	Octet 2 Code d'opération
	<--dis->		Octet 3 Distance
	01bbb110		Octet 4 Code d'opération

Flag : S Z V C
I x I

SET b,reg

Mettre un bit d'un registre

Code d'instruction :	11001011	&CB	Octet 1 Code d'opération
	11bbbrrr		Octet 2 Code d'opération

Flag : S Z V C

SET b,(HL)

Mettre un bit d'une case mémoire

Code d'instruction :	11001011	&CB	Octet 1 Code d'opération
	11bbb110		Octet 2 Code d'opération

Flag : S Z V C

SET b,(XY+dis)

Mettre un bit d'une case mémoire avec adressage indexé

Code d'instruction :	11x11101		Octet 1 Code d'opération
	11001011	&CB	Octet 2 Code d'opération
	<--dis->		Octet 3 Distance
	11bbb110		Octet 4 Code d'opération

Flag : S Z V C

RES b,reg

Annuler un bit d'un registre

Code d'instruction :	11001011	&CB	Octet 1 Code d'opération
	10bbbrrr		Octet 2 Code d'opération

Flag : S Z V C

RES b,(HL)

Annuler un bit d'une case mémoire

Code d'instruction :	11001011	&CB	Octet 1 Code d'opération
	10bbb110		Octet 2 Code d'opération

Flag : S Z V C

RES b,(XY+dis)

Annuler un bit d'une case mémoire avec adressage indexé

Code d'instruction :	11x11101		Octet 1 Code d'opération
	11001011	&CB	Octet 2 Code d'opération
	<--dis->		Octet 3 Distance
	10bbb110		Octet 4 Code d'opération

Flag : S Z V C

CCF

Inverser le bit de retenue

Code d'instruction :	00111111	&3F	Octet 1 Code d'opération

Flag : S Z V C
! ! : inversion

SCF

Mettre le bit de retenue

Code d'instruction :	00110111	&37	Octet 1 Code d'opération

Flag : S Z V C
1

Programmes pour les instructions de bits

Ecrivez un programme qui remplisse l'écran, en MODE 2, de traits d'un point d'épaisseur. Ces traits seront situés au milieu d'un caractère. Vous pouvez réutiliser pour cela la boucle que nous vous donnions dans les programmes précédents.

Listing assembleur

```
A000 2100C0      10  LD  HL,&C000
A003 CBDE        20  SET 3,(HL)
A005 2C          30  INC L
A006 20FB        40  JR  NZ,&A003
A008 24          50  INC H
A009 20F8        60  JR  NZ,&A003
A00B C9          70  RET
```

Programme Basic

```
5 MEMORY &9FFF
10 FOR i=&A000 TO &A00B
20 READ a
30 POKE i,a
40 NEXT i
50 MODE 2
60 CALL &A000
70 END
100 DATA &21,&00,&C0,&CB,&DE,&2C,&20,&FB
110 DATA &24,&20,&F8,&C9
```

SET 3,(HL) peut être remplacé par SET 4,(HL) (Code: &CB,&E6). Dans la ligne de DATA, il faut alors remplacer &DE par &E6.

4.10. SAUTS

Une grande partie des sauts sont des sauts conditionnels, c'est-à-dire qu'ils dépendent de l'état d'un flag. C'est pourquoi nous allons récapituler ici le rôle de chaque flag.
Les flags H et N sont utilisés en arithmétique BCD. Il n'est pas possible d'effectuer de tests sur ces flags. L'état des autres flags (C, P/V, Z, S) peut être testé lors d'un saut conditionnel.

Carry Flag (retenue, C)

Le flag Carry a deux fonctions.

- il indique si une retenue s'est produite lors d'une addition ou d'une soustraction.

- les instructions SRL, SRA, SLA, RR, RL, RRC, RLC, RRA, RLA, RRCA et RLCA utilisent le Carry comme neuvième bit.

Les deux instructions de rotation en BCD, RLD et RRD constituent une exception puisqu'elles ne modifient pas le Carry.

Les instructions logiques AND, OR et XOR annulent toujours le Carry. Elles peuvent être utilisées pour annuler le Carry.

Voici maintenant les autres instructions qui modifient le Carry :

NEG : le flag C est mis si A valait 0 avant exécution de l'instruction.

DAA : l'effet de cette instruction est complexe. Comme nous n'avons pas traité les instructions arithmétiques en BCD, nous ne décrirons pas non plus l'effet de cette instruction sur le Carry.

SCF : Set Carry Flag
Cette instruction met le Carry à 1.

CCF : Complement Carry Flag
Cette instruction inverse le Carry

Les autres instructions ne modifient pas le Carry !

Parity/Overflow (parité/dépassement- P/V)

Ce flag a différentes fonctions suivant l'instruction exécutée.

- *Dépassement*

Pour les instructions arithmétiques sur 8 bits ADD, ADC, SUB, pour les instructions sur 8 bits INC, DEC, NEG et CP, ce flag indique un dépassement. Cela signifie que le signe d'un nombre a été modifié de façon incorrecte.

Attention : les instructions ADD, INC et DEC sur 16 bits ne modifient pas le flag V

- *Parité*

Ce flag est utilisé comme flag P pour les instructions d'entrée (IN), de rotation et de décalage RR, RL, RRC, RLC, RLD, RRD, SLA, SRA et SRL, pour les instructions logiques AND, OR, XOR et pour DAA. P est mis lorsque le nombre de 1 dans un octet est pair et il est annulé lorsque le nombre de bits mis est impair.

Attention : les instructions RLA, RRA, RLCA, RRCA ne modifient pas P!

- pour les instructions de bloc LDD, LDI, CPD, CPI et CPIR, P/V est annulé lorsque BC était égal à 0 (BC est le registre de comptage), sinon il est mis.

LDDR et LDIR ont pour cette raison pour effet de toujours annuler P/V.

- *Interrupt Flag*

Pour les instruction LD A,I et LD A,R, le flag P/V reçoit toujours la valeur des flip-flops d'interruption (IFF). Celle-ci est 0 lorsque les interruptions masquables sont inhibées et 1 lorsqu'elles sont autorisées.

Attention : l'instruction BIT ainsi que toutes les instructions d'entrée et sortie de bloc fixent ce flag de façon arbitraire. Elles peuvent donc dans certains cas modifier la valeur antérieure. Les autres instructions ne modifient pas ce flag.

Zero Flag (Zéro, Z)

Le flag Z indique si la valeur d'un octet est zéro. Si c'est le cas, le flag est mis, si ce n'est pas le cas, il est annulé.

Pour les instructions de comparaison, Z est mis sur 1 s'il y a égalité et il est annulé dans le cas contraire.

Pour l'instruction BIT, le flag Zéro est mis sur 1 si le bit testé vaut 0 et il est annulé dans le cas contraire.

Les instructions suivantes modifient le flag Z :

arithmétiques :	ADD, ADC, SUB, SBC, INC, DEC, NEG, DAA
Attention :	ADD, INC et DEC 16 bits: pas de modification !
comparaison :	CP: Z=1 si égalité, sinon Z=0
bit :	BIT
rotation/décalage :	RR, RL, RRC, RLC, SRL, SRA, SLA, RLD, RRD
Attention :	RRA, RLA, RRCA, RLCA: pas de modification !

Sign Flag (signe, S)

Le flag Signe contient la valeur du bit 7 d'un octet. Ce bit correspond en effet au signe en arithmétique signée.

Les instructions suivantes modifient le flag S :

Toutes les instructions arithmétiques et logiques :

ADD, ADC, SUB, SBC, INC, DEC, NEG, DAA, AND, OR, XOR, CP

Les instructions de rotation et de décalage :

RR, RL, RRC, RLC, SRL, SRA, SLA, RLD, RRD

Les instructions de recherche dans bloc :

CPD, CPI, CPDR, CPIR

Les instructions d'entrée IN et de chargement LD A,I et LD A,R

Attention : ADD 16 bits, INC 16 bits et DEC 16 bits ainsi que RLA, RRA, RLCA, RRCA : aucune modification !

L'instruction BIT et les instructions d'entrée et de sortie de bloc INI, IND, OUTI, OUTD, INIR, INDR, OTIR, OTDR ont une influence indéterminée sur l'état du flag S.

Vous trouverez en annexe l'effet sur les flags des différentes instructions.

Le Z80 connaît 5 types de sauts différents :

- sauts à l'intérieur du programme principal qui correspondent à l'instruction Basic GOTO.
- sauts de sous-programmes (CALL et RET) qui correspondent aux instructions Basic GOSUB et RETURN.
- sauts relatifs (JUMP RELATIVE) qui ressemblent à l'instruction Basic FOR NEXT.
- instructions de RESTART (RST) qui exécutent un branchement à une adresse fixée d'avance. L'instruction RST n'a pas d'équivalent en Basic.
- sauts d'interruption (voyez les instructions de commande).

Les trois premiers types de saut existent sur le Z80 sous forme de sauts inconditionnels ou conditionnels, c'est-à-dire dépendant de l'état d'un flag. Pour les sauts conditionnels le saut peut s'effectuer en fonction de l'état des flags Z, C, P/V et S. Il est possible de tester pour chaque flag s'il est bien à 0 ou s'il est bien à 1.

En langage assembleur, on utilise les abréviations suivantes :

Z =	sauter si Zéro	(Z=1)
NZ=	sauter si pas Zéro	(Z=0)
C =	sauter si retenue	(C=1)
NC=	sauter si pas retenue	(C=0)
PO=	sauter si pas parité	(P/V=0)
PE=	sauter si parité	(P/V=1)
P =	sauter si plus (+)	(S=0)
M =	sauter si moins (-)	(S=1)

Le Z80 dispose en outre d'une instruction spéciale de boucle qui décrémente le registre B et exécute ensuite un saut relatif, tant que B<>0. Cette instruction s'appelle DJNZ (Decrement Jump Non Zero).

JUMP

Les sauts à l'intérieur du programme principal (actuel! le programme principal peut être lui-même un sous-programme d'un autre programme principal de niveau supérieur) sont exécutés avec l'instruction JP. L'adresse de saut peut être indiquée grâce à deux modes d'adressage différents.

Adressage absolu :

```
JP adr    ou  JP cond,adr
```

cond est mis pour une condition, c'est-à-dire pour Z, NZ, C, NC, PO, PE, P ou M.

JP adr	-	saute INCONDITIONNELLEMENT à l'adresse indiquée
JP cond,adr	-	saute à l'adresse indiquée si la CONDITION est remplie, si la condition n'est pas remplie, c'est l'instruction suivante qui est exécutée

Comparaison

JP adr	Basic: GOTO numéro de ligne
JP NZ,adr	Basic: IF Z=0 THEN GOTO numéro de ligne
JP Z,adr	Basic: IF Z=1 THEN GOTO numéro de ligne

Le processeur exécute un saut en plaçant l'adresse indiquée dans le compteur de programme (PC). C'est alors la première instruction figurant à cette adresse qui est lue et exécutée.

Pour l'adressage absolu, le code d'opération sur 1 octet est suivie de l'adresse de saut voulue, dans le format octet faible, octet fort. Comme les instructions sur trois octets sont relativement lentes, il existe également des instructions de saut relatif qui ne nécessitent que 2 octets. Les sauts avec adressage indirect ont un code d'opération sur 1 octet.

Adressage indirect

Format :

JP (X)

X: HL, IX ou IY

JP (X) saute à l'adresse indiquée par le registre X.

CALL/RET

Nous avons déjà indiqué comment les instructions CALL et RET stockent ou lisent l'adresse de retour d'un sous-programme, à l'aide de la pile et du pointeur de pile (SP). L'appel d'un sous-programme peut être conditionnel ou inconditionnel. L'adresse de saut (= l'adresse de début du sous-programme) est indiquée de manière absolue.

Format :

```
CALL adr ou   CALL cond,adr
```

Lors de l'exécution de cette instruction, toutes les manipulations nécessaires de la pile, du SP et du PC sont exécutées automatiquement :

Après avoir lu entièrement l'instruction, le PC indique l'adresse de l'instruction suivante. C'est alors qu'interviennent les opérations suivantes :

```
(SP-1)=(octet fort)  PC
(SP-2)=(octet faible) PC
SP=SP-2
PC=adr
```

L'instruction suivante est lue à l'adresse indiquée par PC. Un sous-programme doit être terminé par l'instruction RET. Le RETURN peut également être inconditionnel ou conditionnel.

Format :

```
RET        ou   RET cond
```

Les opérations suivantes se déroulent lors de l'exécution de l'instruction :

```
(octet faible) PC=(SP)
(octet fort) PC=(SP+1)
SP=SP+2
```

L'exécution du programme se poursuit à l'adresse retirée de la pile.

Au contraire de l'instruction CALL, l'instruction RET n'est longue que d'1 octet. Pour CALL, l'adresse 16 bits doit être indiquée et cette instruction est donc longue de 3 octets.

Il existe deux instructions de retour spéciales RETI et RETN qui sont traitées dans le chapitre sur les instructions de commande.

Jump Relative

Les instructions de saut relatif effectuent un saut relativement à l'adresse actuelle. C'est donc la distance qui doit être indiquée. Le premier octet est le code d'opération et le second indique la distance avec son signe (en complément à deux). Cette procédure s'appelle l'adressage relatif. Dans ce cas, la distance est également appelée offset.

Format :

```
JR e   ou   JR cond,e
    e: offset
cond: Z, NZ, C, NC
```

Les sauts relatifs conditionnels ne peuvent se faire qu'en fonction des flags C et Z.

Mais comment est calculée la distance pour un saut conditionnel ?

Considérons le dernier programme du chapitre 4.9. A l'adresse &A006 figure l'instruction JR. Le but du saut est l'instruction SET 3,(HL) à l'adresse &A003. La différence entre l'adresse actuelle et l'adresse objet est donc de &A006 - &A003 = 3. Mais comme il s'agit d'un saut "en arrière" (l'adresse objet est inférieure à l'adresse actuelle), la distance ou offset est de -3. Pour calculer quel sera le deuxième octet de notre instruction, il nous faut oter 2 de la distance. Pourquoi cette soustraction ?

Le processeur commence toujours par lire une instruction entièrement. En l'occurrence, il lira donc le code d'opération (octet 1) et la distance (octet 2). Chaque fois que le PC lit un octet, il est incrémenté ensuite de 1. Une fois donc que l'instruction a été lue entièrement, le compteur de programme indique donc l'adresse de la prochaine instruction. Le compteur de programme indique donc une adresse qui est plus élevée de 2 que l'adresse de l'instruction de saut. Le Z80 effectue le saut en

ajoutant au PC la distance. C'est pourquoi il nous faut donc tenir compte du fait que le PC a été augmenté de 2 lors de la lecture de l'instruction. Lors d'un saut "en arrière", il est en effet nécessaire de sauter également au-dessus de ces deux octets. La distance à fournir se calcule donc ainsi :

```
-3-2=-5= &FB en complément à deux
```

En langage assembleur, cette différence 2 ne doit pas être indiquée. L'instruction est donc JR $-3 ($ est mis pour adresse actuelle de l'instruction). Le programme d'assembleur effectue la soustraction de 2 ainsi que la conversion en complément à 2. Il est également possible d'indiquer l'adresse absolue, soit &A003. Le programme d'assembleur calculera alors la différence entre $ (adresse actuelle) et &A003 et stockera la distance correcte. Bien que l'adresse 16 bits puisse être indiquée dans l'instruction en assembleur, l'instruction en langage-machine est bien une instruction de saut relatif. Si l'on prend en compte les deux octets de l'instruction, les sauts peuvent se faire sur une distance maximale de +129 ou -126 relativement à l'adresse actuelle.

Récapitulons le mode de calcul de l'octet de distance :

l'instruction de saut figure à l'adresse ADR
l'adresse objet figure à l'adresse ADRO
distance = ADRO - ADR
octet à stocker : (distance - 2) en complément à 2

Exercice :

Dans le listing assembleur (chapitre 4.9) figure à l'adresse &A009 un saut relatif. Le but du saut est toujours &A003. Calculez l'octet de distance et comparez le résultat auquel vous parvenez avec le listing assembleur.

Nous avons maintenant traité les principales instructions. Revenons sur un programme qui figurait dans le chapitre sur les instructions de chargement.

La tâche du programme consistait à remplir l'angle supérieur gauche de l'écran. Ce problème peut être mieux résolu par une boucle.

En Basic nous aurions :

```
10 FOR I=&C000 TO &FFFF STEP &800
20 POKE I,&FF
30 NEXT
```

Pour traduire ce programme en langage-machine, nous chargeons l'adresse de début &C000 dans la paire de registres HL. Pour traduire l'instruction STEP &800, &800 sera chargé dans DE et une addition sur 16 bits sera exécutée. Si le Carry est mis après l'addition, c'est que le programme est terminé :

Solution :

```
A000 2100C0      10    LD   HL,&C000
A003 110008      20    LD   DE,&800
A006 36FF        30    LD   (HL),&FF
A008 19          40    ADDHL,DE
A009 30FB        50    JR   NC,&A006
A00B C9          60    RET
```

Modifiez maintenant ce programme de façon à ce que la case ne soit plus remplie mais que le caractère qui s'y trouve soit représenté en inversion vidéo.

Solution :

```
A000 2100C0      10    LD   HL,&C000
A003 110008      20    LD   DE,&800
A006 7E          30    LD   A,(HL)
A007 2F          40    CPL
A008 77          50    LD   (HL),A
A009 19          60    ADDHL,DE
A00A 30FA        70    JR   NC,&A006
A00C C9          80    RET
```

L'inversion de l'octet peut bien sûr être opérée également avec l'instruction XOR &FF au lieu de CPL. Cette instruction est cependant plus longue (2 octets) et donc plus lente que CPL.

L'instruction permet une programmation plus aisée des boucles. La distance est indiquée comme second octet, comme pour JR. Le registre B est utilisé comme compteur. Pour que la boucle soit parcourue 8 fois, il faut charger 8 dans le compteur B puisque dès que B=0 le saut n'est plus exécuté. L'instruction JR est remplacée par DJNZ et 8 est chargé au début dans B.

Listing assembleur

```
A000 0608      10   LD  B,8
A002 2100C0    20   LD  HL,&C000
A005 110008    30   LD  DE,&800
A008 7E        40   LD  A,(HL)
A009 2F        50   CPL
A00A 77        60   LD  (HL),A
A00B 19        70   ADDHL,DE
A00C 10FA      80   DJNZ   &A008
A00E C9        80   RET
```

Restart

Ce type d'instructions de saut a la longueur minimum d'un octet. Ce sont donc les instructions de saut qui sont exécutées le plus rapidement, si l'on excepte RET. L'instruction RST ou Restart provoque un saut à un sous-programme situé à une adresse figurant dans la partie inférieure de la mémoire. Il y a 8 instructions Restart. Les adresses auxquelles sautent ces instructions sont &0, &8, &10, &18, &20, &28, &30 et &38.

Format :

```
RST adr
adr: une des huit adresses indiquées ci-dessus.
```

Comme l'instruction Restart est l'instruction de saut la plus rapide, ce sont des routines ou des sauts à des routines importantes et souvent utilisées qui figurent dans la partie inférieure de la mémoire. Nous étudierons plus loin la fonction exacte des différentes instructions Restart.

Liste d'instructions

JP adr

Saut inconditionnel

PC=adr

Code d'instruction :	11000011	&C3	Octet 1 Code d'opération
	<--adr->		Octet 2 Adresse
	<--adr->		Octet 3 Adresse

Flag : S Z V C

JP cond,adr

Saut conditionnel, si cond est remplie

PC=adr

Code d'instruction :	11ccc010	Octet 1 Code d'opération
	<--adr->	Octet 2 Adresse
	<--adr->	Octet 3 Adresse

Flag : S Z V C

JR of

Saut relatif inconditionnel (of = offset -distance-)

PC=PC+of

Code d'instruction :	00011000	&18	Octet 1 Code d'opération
	<-of-2->		Octet 2 Offset

Flag : S Z V C

JR cond,of

Saut relatif conditionnel, si cond.

PC=PC+of

Code d'instruction :	001cc000		Octet 1 Code d'opération
	<-of-2->		Octet 2 Offset

Flag : S Z V C

JP (HL)

Saut à travers le contenu du registre

PC=HL

Code d'instruction :	11101001	&E9	Octet 1 Code d'opération

Flag : S Z V C

JP (XY)

Saut à travers le contenu du registre d'index

PC=XY

Code d'instruction :	11x11101		Octet 1 Code d'opération
	11101001	&E9	Octet 2 Code d'opération

Flag : S Z V C

DJNZ of

Instruction de boucle

PC=XY

Code d'instruction :	00010000	&10	Octet 1 Code d'opération
	<-of-2->		Octet 2 Offset

Flag : S Z V C

CALL adr

Appel d'un sous-programme

(SP-1)=PC High,(SP-2)=PC Low,PC=adr

Code d'instruction :	11001101	&CD	Octet 1 Code d'opération
	<--adr->		Octet 2 Adresse
	<--adr->		Octet 3 Adresse

Flag : S Z V C

CALL cond,adr

Appel conditionnel d'un sous-programme

(SP-1)=PC High,(SP-2)=PC Low,PC=adr

Code d'instruction :	11ccc100	Octet 1 Code d'opération
	<--adr->	Octet 2 Adresse
	<--adr->	Octet 3 Adresse

Flag : S Z V C

RET

Retour d'un sous-programme

PC Low=(SP),PC High=(SP+1)

Code d'instruction : 11001001 &C9 Octet 1 Code d'opération

Flag : S Z V C

RET cond

Retour conditionnel d'un sous-programme

PC Low=(SP),PC High=(SP+1)

Code d'instruction : 11ccc000 Octet 1 Code d'opération

Flag : S Z V C

RETI

Retour du programme de service des interruptions

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	01001101	&4D	Octet 2 Code d'opération

Flag : S Z V C

RST p

Saut à des routines de service (p = une des adresses &00, &08, &10, &18, &20, &28, &30, &38)

(SP-1)=PC High,(SP-2)=PC Low,PC High=0, PC Low=p

Code d'instruction : 11ttt111 Octet 1 Code d'opération

Flag : S Z V C

ttt :	&00 - 000	&20 - 100
	&08 - 001	&28 - 101
	&10 - 101	&30 - 110
	&18 - 011	&38 - 111

4.11. INSTRUCTIONS DE COMMANDE

Les instructions de commande agissent sur le mode de travail ou le déroulement des opérations dans l'unité centrale.

L'instruction NOP

NOP est mis pour No OPeration. NOP est donc une instruction sans fonction. Il ne s'agit pourtant pas du tout d'une instruction inutile. Cette instruction "qui ne fait rien" peut en effet être

utilisée pour ralentir volontairement un programme (une instruction NOP dure une microseconde sur le CPC (10^-6 secondes). Cette instruction peut par ailleurs être utilisée lors du développement des programmes pour réserver de la place, ce qui peut faciliter la recherche et surtout la correction des erreurs. Le code d'opération de NOP est &00. Comme sur le CPC toute zone dans laquelle rien n'a été encore écrit est remplie de zéros, si le programme exécute par erreur une zone ou ne figure aucun programme, rien ne sera détruit ou modifié puisque NOP ne provoque aucune action.

L'instruction HALT

Cette instruction interrompt les opérations de l'unité centrale jusqu'à ce qu'un Reset ou une interruption se produisent.

Instructions de commande des interruptions

Les interruptions servent essentiellement au traitement de tâches importantes dans l'ordinateur. Une interruption est un message envoyé par un composant, par lequel ce dernier signale l'intervention d'un état, comme par exemple pour l'attente d'une entrée par un périphérique d'entrée/sortie. Ces messages sont traités par l'unité centrale selon leur importance respective. Un programme normal en cours est interrompu par une interruption. Les interruptions jouent un rôle important dans les opérations d'entrée/sortie. L'Amstrad vous donne également la possibilité de programmer les interruptions à partir du Basic (EVERY-AFTER). Pour ces instructions, l'interruption est déclenchée par l'horloge interne du processeur. Si une interruption autorisée est demandée, le programme saute à l'adresse de début d'un sous-programme qui exécute les actions correspondant aux différentes interruptions. On quitte ce programme de service des interruptions pour retourner au programme principal avec RETI (RETurn from Interrupt).

On distingue entre les interruptions masquables et non-masquables. Les dernières sont exécutées en toutes circonstances. Elles ont le plus haut rang de priorité. Le retour d'une interruption non-masquable se fait avec RETN.

DI-Disable Interrupt et EI Enable Interrupt

L'instruction DI a pour effet qu'à partir de son exécution les interruptions masquables seront ignorées. Les interruptions seront inhibées jusqu'à ce qu'elles soient à nouveau autorisées par EI (Enable Interrupt).
Le Z80 dispose de trois modes d'interruption: IM0, IM1 et IM2

IM 0 (Interrupt Modus 0)

IM 0 permet de passer du mode standard 1 au mode 0.
Après une interruption, le processeur attend dans ce mode une instruction d'un périphérique externe.

IM 1

C'est le mode standard dans lequel on se trouve après la mise sous tension de l'ordinateur.
Dans ce mode on saute automatiquement à une adresse fixée.

IM 2 (Interruption vecteur)

En mode IM 2, on saute à une adresse figurant dans une table, en fonction de l'interruption.

Liste d'instructions

NOP

Instruction sans fonction

Code d'instruction : 00000000 &00 Octet 1 Code d'opération

Flag : S Z V C

HALT

Placer l'unité centrale en état HALT.

Code d'instruction : 01110110 &76 Octet 1 Code d'opération

Flag : S Z V C

DI

Inhiber les interruptions

IFF=0

Code d'instruction : 11110011 &F3 Octet 1 Code d'opération

Flag : S Z V C

EI

Autoriser les interruptions

IFF=1

Code d'instruction : 11111011 Octet 1 Code d'opération

Flag : S Z V C

IM 0

Fixer le mode d'exécution des interruptions.

Code d'instruction : 11101101 &ED Octet 1 Code d'opération
01000110 &46 Octet 2 Code d'opération

Flag : S Z V C

IM 1

Fixer le mode d'exécution des interruptions.

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	01010110	&56	Octet 2 Code d'opération

Flag : S Z V C

IM 2

Fixer le mode d'exécution des interruptions.

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	01011110	&5E	Octet 2 Code d'opération

Flag : S Z V C

RETN

Retour du programme de service des interruptions NMI.

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	01000101	&45	Octet 2 Code d'opération

Flag : S Z V C

RETI

Retour du programme de service des interruptions.

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	01001101	&4D	Octet 2 Offset

Flag : S Z V C

4.12. INSTRUCTIONS D'ENTREE/SORTIE

Toutes les unités centrales ne disposent pas d'instructions spéciales d'entrée/sortie. Leur utilisation facilite notablement la programmation des composants d'entrée/sortie. Ces instructions sont réellement complexes et nous n'en dirons pas plus sur ce sujet.

Liste d'instructions

IN A,(donnée)

Instruction d'entrée

A=(donnée)

Code d'instruction :			
	11011011	&DB	Octet 1 Code d'opération
	<--co-->		Octet 2 Constante

Flag : S Z V C

IN reg,(C)

Instruction d'entrée

reg=(C)

Code d'instruction :			
	11101101	&ED	Octet 1 Code d'opération
	01rrr000		Octet 2 Registre

Flag :	S	Z	V	C
	x	x	x	

INI

Instruction d'entrée de bloc

(HL)=(C),B=B-1,HL=HL+1

Code d'instruction :	11101101	&ED	Octet 1	Code d'opération
	10100010	&A2	Octet 2	Code d'opération

Flag : S Z V C
U x U

INIR

Instruction d'entrée de bloc

(HL)=(C),B=B-1,HL=HL+1

Code d'instruction :	11101101	&ED	Octet 1	Code d'opération
	10110010	&B2	Octet 2	Code d'opération

Flag : S Z V C
U 1 U

IND

Instruction d'entrée de bloc

(HL)=(C),B=B-1,HL=HL-1

Code d'instruction :	11101101	&ED	Octet 1	Code d'opération
	10101010	&AA	Octet 2	Code d'opération

Flag : S Z V C
U x U

INDR

Instruction d'entrée de bloc

(HL)=(C),B=B-1,HL=HL-1

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	10111010	&BA	Octet 2 Code d'opération

Flag : S Z V C
U 1 U

OUT (donnée),A

Instruction de sortie

(donnée)=A

Code d'instruction :	11010011	&D3	Octet 1 Code d'opération
	<--co-->		Octet 2 Constante

Flag : S Z V C

OUT (C),reg

Instruction de sortie

(C)=reg

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	01rrr001		Octet 2 Registre

Flag : S Z V C

OUTI

Instruction de sortie de bloc

(C)=(HL),B=B-1,HL=HL+1

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	10100011	&A3	Octet 2 Code d'opération

Flag : S Z V C
U x U

OTIR

Instruction de sortie de bloc

(C)=(HL),B=B-1,HL=HL+1

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	10110011	&B3	Octet 2 Code d'opération

Flag : S Z V C
U 1 U

OUTD

Instruction de sortie de bloc

(C)=(HL),B=B-1,HL=HL-1

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	10101011	&AB	Octet 2 Code d'opération

Flag : S Z V C
U x U

OTDR

Instruction de sortie de bloc

(C)=(HL),B=B-1,HL=HL-1

Code d'instruction :	11101101	&ED	Octet 1 Code d'opération
	10111011	&BB	Octet 2 Code d'opération

Flag : S Z V C
U 1 U

5. Programmation du Z80

5.1. L'ASSEMBLEUR

Pour que nous n'ayons plus à transcrire les prochains programmes en langage-machine à la main, nous avons écrit un assembleur Z80. L'assembleur produit le code machine (code ou programme OBJET) correspondant à un programme écrit en langage assembleur (programme SOURCE). Il calcule par exemple automatiquement les distances pour les instructions de saut relatif. Nous pouvons ainsi nous dispenser du pénible travail que constituent la traduction manuelle du programme, la recherche des codes d'opération, etc...

Certaines conventions sont admises pour les programmes d'assembleur Z80.

Une ligne d'assembleur se présente ainsi :

Label Instruction Opérande ;Commentaire

Comme nous voulons utiliser l'éditeur Basic pour entrer nos programmes, chaque ligne d'assembleur doit recevoir un numéro de ligne.

Nous allons maintenant définir le format d'entrée de l'assembleur. Pour éviter des erreurs dans l'utilisation de l'assembleur, les explications suivantes sont très importantes et nous vous demandons donc de les étudier avec beaucoup d'attention.

Label (étiquette) :

Une ligne peut commencer par un label (après le numéro de ligne bien sûr). Un label est une variable dont le nom ne doit pas comporter plus de 6 caractères. Les noms de label doivent commencer par une lettre. Les instructions assembleur ne doivent pas être utilisées comme nom de label.

L'utilisation de labels facilite notamment la programmation des sauts :

```
        .
        .
ANF     Instruction     ; ANF  : Label
        .
        .
        JR ANF          ;      : sauter à ANF
```

L'assembleur calcule automatiquement la distance correcte.

Instruction (Mnémonique) :

Après la présence éventuelle du label vient l'instruction. Label et instruction doivent être séparés entre eux par un espace. La mnémonique doit être une instruction assembleur correcte. Ces instructions sont celles que nous avons présentées dans toutes nos listes d'instructions: LD, ADD, INC etc...

Opérande :

Le mot instruction est suivi, toujours séparé par un espace, de l'opérande. Pour les sauts, l'adresse de saut peut être fournie sous forme d'un label. Il faut bien sûr alors que ce label existe.

Les labels peuvent également remplacer les constantes ou les distances.

Il ne doit jamais y avoir d'espaces à l'intérieur de l'opérande !

Commentaire :

En fin de ligne, séparé par un espace et un point-virgule, peut figurer un commentaire. Tout texte placé à la suite d'un point-virgule est ignoré lors de l'assemblage (=la traduction du programme assembleur en code machine). Les commentaires sont une aide précieuse pour une meilleure compréhension ultérieure des programmes.

Lors de l'assemblage, l'assembleur produit un listing assembleur qui peut être sorti sur imprimante ou à l'écran. Le code produit peut d'autre part être stocké sur cassette.

Le listing assembleur a la structure suivante :

&adr	&codes	No lgn BASIC	Label	Instruction	Opérande	;Commentaire
A003	36CC	50	SUITE	LD	(hl),bitmat	;matrice bits
A005	23	60		INC	hl	;augmenter hl

Outre les instructions Z80, l'assembleur connaît toute une série de pseudo-instructions. Il s'agit d'instructions à l'assembleur comme par exemple END qui signifie à l'assembleur qu'il n'a plus à chercher d'autres instructions et qu'il doit mettre fin à l'assemblage.

EQU est une autre instruction importante (en anglais equal=égal). EQU permet de définir la valeur d'une variable.

Nom de variable EQU Valeur

L'instruction ORG (ORIGINE) indique à partir de quelle adresse le programme doit être stocké. Nous utiliserons le plus souvent &A000 comme adresse de départ.

Pour l'indication des nombres, on admet les conventions suivantes :

Les nombres hexadécimaux doivent être précédés de "&".

Les nombres binaires doivent être précédés de "&X".

Un nombre qui n'est précédé d'aucun des deux signes sera interprété comme un nombre décimal.

Les conventions standard pour les assembleurs Z80 sont un H placé à la suite d'un nombre hexadécimal et un B placé à la suite d'un nombre binaire. Nous avons cependant préféré adopter la convention ci-dessus qui correspond au Basic du CPC.

Essayez maintenant notre assembleur en entrant simplement un petit programme.

Le programme source assembleur peut être entré indépendamment du programme d'assembleur. Le premier programme du chapitre 1.2 se présenterait donc maintenant ainsi :

```
 10 ' org &a000
 20 ' bildad equ &c000 ;début mémoire écran
 30 ' bilmat equ &cc ;matrice point écran
 40 ' ld hl,bildad
 50 ' suite ld(hl),bitmat
 60 ' inc hl
 70 ' cp h ;comparer avec 0
 80 ' jr nz,suite
 90 ' ret
100 ' end
```

Lors de l'entrée du programme, vous pouvez utiliser indifféremment l'écriture en majuscules ou en minuscules.

Notez que chaque numéro de ligne doit être suivi d'un espace et du caractère apostrophe "'". Si vous oubliez ce caractère, la ligne correspondante ne pourra pas être traduite plus tard par l'assembleur et vous obtiendrez le message d'erreur :

"Erreur ' missing in ..."

La ligne 10 fixe la zone de stockage du programme à partir de &A000.

La ligne 20 affecte à la variable Bildad (V) la valeur &C000.

Au lieu de &C000, il sera ensuite toujours possible d'écrire "Bildad" (V). Une utilisation judicieuse des variables rend un programme plus clair. Dans la boucle du programme, nous avons utilisé le label "suite" comme but du saut. Pour le reste, nous avons utilisé les instructions assembleur normales.

Si un commentaire suit dans une ligne, il est séparé par un point-virgule. Il est important de noter qu'un espace doit en outre précéder ce point-virgule. Les espaces marquent pour l'assembleur la séparation entre les différents champs d'une ligne assembleur (label, instruction, opérande et commentaire). Ces différents champs doivent donc toujours être séparés entre eux par des espaces mais il ne doit bien sûr pas y avoir d'espace à l'intérieur d'un champ.

Exemple :

```
( HL )  INCORRECT ! ! !
(HL)    CORRECT ! ! !
```

A la fin du programme doit figurer la pseudo-instruction END. Cette instruction signifie pour l'assembleur que l'assemblage doit être terminé ici.

Sauvegardez le programme entré avec >SAVE"Nom"< et chargez à sa suite l'assembleur avec >MERGE<. L'assembleur occupe les numéros de ligne 10000 et suivants. Les numéros de ligne utilisés pour les programmes source doivent donc être inférieurs.

REMARQUE pour l'utilisateur du lecteur de disquette : un programme qui doit être chargé à partir de la disquette avec >MERGE< doit être un fichier ASCII sans tête de fichier (header). Vous pouvez réaliser cela en plaçant à la suite de l'instruction >SAVE"Nom"< du programme source ',A'. Comme le chargement de ce type de fichier est assez long, nous vous recommandons de toujours charger d'abord l'assembleur (comme un programme Basic normal) puis le programme source (sous forme d'un fichier ASCII sans tête de fichier. Comme AMSDOS occupe environ 500 octets en mémoire Ram, qui sont affectés de façon dynamique, nous recommandons aux utilisateurs de disquette de stocker leurs programmes en langage-machine au maximum jusqu'à l'adresse &A600 pour éviter toutes complications.

Vous pouvez maintenant lancer l'assemblage simplement en entrant >RUN<.

L'assembleur vous demande le nom du programme, si vous souhaitez avoir un listing assembleur de l'assemblage et si ce listing doit être imprimé. Vous pouvez accepter les réponses proposées (o pour listing et n pour imprimante en appuyant sur >ENTER<. Choisissez d'abord les entrées standard.

C'est maintenant que commence l'assemblage proprement dit.

Le listing assembleur que vous connaissez est sorti sur l'écran. Si une erreur se produit, les messages correspondants sont sortis devant la ligne concernée. A la fin du listing sont indiqués, s'il y a lieu, les labels et variables non définies. Ensuite viennent le nom du programme, l'adresse de début, l'adresse de fin, la longueur du programme et le nombre d'erreurs. Si des erreurs se sont produites, elles peuvent être corrigées dans la ligne Basic correspondante.

A la fin du listing, une table de tous les labels avec leurs valeurs respectives est sortie. Les labels sont classés dans l'ordre de leur apparition. On vous demande enfin si le code machine doit être stocké.

Si vous entrez "o" le code produit est sauvegardé comme un fichier binaire sous le nom entré avec le suffixe ".OBJ" (=code OBJet). Après l'assemblage, le programme machine se trouve en mémoire à l'adresse que vous aviez indiquée et il peut être appelé avec >CALL<.

Si vous voulez assembler d'autres programmes assembleur, l'ancien programme source peut être supprimé avec l'instruction >DELETE 2-9999<. Vous pouvez alors charger le nouveau programme avec >MERGE<. Voici en exemple le listing assembleur complet de notre programme :

```
A000           10            ORG   &a000
A000           20   BILDAD   EQU   &c000  ;debut memoire
A000           30   BITMAN   EQU   &cc  ;matrice point-ecran
A000   2100C0  40            LD    hl,bildad
A003   36CC    50   SUITE    LD    (hl),bitman
A005   23      60            INC   hl
```

```
A006  BC    70    CP   h ;comparer avec 0
A007  20FA  80    JR   nz,suite
A009  C9    90    RET
End Assumed

Programme : ecran
Debut : &A000   Fin : &A009
Longueur : 000A
  0   Erreur
Table de variables :
BILDAD  C000  BITMAT  OOCC  SUITE  A003
```

Essayez en tapant le listing du programme d'assembleur de comprendre la structure fondamentale de l'assembleur en vous aidant des explications que nous vous donnons à la suite du listing.

ATTENTION : ne modifiez jamais la ligne 1. Veillez notamment à ne pas rajouter d'espaces, même à la fin de la ligne. D'une manière générale, n'apportez aucune modification aux lignes jusqu'à 10010 incluses ainsi qu'au début de la partie d'initialisation en lignes 14160-14180. Les valeurs de départ de bpc (V) et de vapt (V) risqueraient alors de comporter des valeurs incorrectes et le programme ne fonctionnerait plus.

```
1 MEMORY &9FFF:GOTO 10000
10000 REM ******** Assembleur  Z80 ********
10001 REM **** c 1984 by Holger Dullin ****
10010 GOTO 14160
10020 LOCATE 20,4:PRINT"A s s e m b l e u r   Z 8 0"
10030 LOCATE 5,8:INPUT"Nom du programme :",nom$
10040 LOCATE 19,11:PRINT"o"
10050 LOCATE 5,11:INPUT"Listing (o/n):",t$
10060 IF t$="n" THEN listflag=0:GOTO 10100 ELSE listflag=-1
10070 LOCATE 22,13:PRINT"n";
10080 LOCATE 5.13:INPUT"Imprimante (o/n):",t$
10090 IF t$="o" THEN sor=8:PRINT#sor ELSE sor=0
10100 REM Start Assembly **********
10110 MODE 2
10120 REM Tester debut de ligne --------
```

```
10130 laze=FNdeek(bpc)
10140 bpc=bpc+2
10150 zenr=FNdeek(bpc)
10160 IF zenr>9999 THEN PRINT#sor,"End Assumed":GOTO 13400
10170 bpc=bpc+2
10180 IF FNdeek(bpc)<>49153 THEN PRINT#sor,"Erreur ' missing in";zenr:
bpc=bpc+laze-4:feza=feza+1:GOTO 10130
10190 bpc=bpc+2
10200 REM lire ligne ------------------
10210 POKE vapt,laze-7
10220 POKE vapt+1,bpc-256*INT(bpc/256)
10230 POKE vapt+2,INT(bpc/256)
10240 REM decomposer ligne ------------
10250 zeia$=zei$
10260 bpc=bpc+laze-6
10270 FOR i=0 TO 3:a$(i)="":NEXT
10280 bepo=INSTR(zei$,";")
10290 IF bepo=0 THEN bemer$="":GOTO 10320
10300 bemer$=RIGHT$(zei$,LEN(zei$)-bepo+1)
10310 zei$=LEFT$(zei$,bepo-1)
10320 j=0
10330 IF LEFT$(zei$,1)=" " THEN zei$=RIGHT$(zei$,LEN(zei$)-1):GOTO 103
30
10340 sppo=INSTR(zei$," ")
10350 IF zei$="" THEN j=j-1:GOTO 10420
10360 IF sppo=0 THEN 10410
10370 a$(j)=LEFT$(zei$,sppo-1):zei$=RIGHT$(zei$,LEN(zei$)-sppo)
10380 IF zei$="" THEN j=j-1:GOTO 10420
10390 IF j>3 THEN 10420
10400 j=j+1:GOTO 10330
10410 a$(j)=zei$
10420 IF j>2 THEN 13250
10430 REM Interpretation --------------
10440 j=0
10450 bef$=LEFT$(UPPER$(a$(j))+"     ",4)
10460 po=INSTR(teadr$,bef$)
10470 IF po<>0 THEN lp=0:GOTO 11190
10480 po=INSTR(teb1$,bef$)
```

```
10490 IF po<>0 THEN lp=1:GOTO 10810
10500 po=INSTR(teed$,bef$)
10510 IF po<>0 THEN lp=2:pw(1)=&ED:GOTO 10850
10520 REM test pseudo instr -----------
10530 po=INSTR(teps$,bef$)
10540 IF po<>0 THEN 10890
10550 REM a$(j)=label ?  --------------
10560 IF j>0 THEN 13250
10570 IF a$(0)="" THEN 13100
10580 a$=a$(0)
10590 GOSUB 13630
10600 IF nolafl THEN 13280
10610 label$=UPPER$(lab$)
10620 wert=mpc
10630 lata$(ltp)=label$:wita(ltp)=mpc:ltp=ltp+1
10640 FOR i=0 TO ultp:IF label$=ulata$(i) THEN 10670
10650 NEXT i
10660 j=j+1:GOTO 10450
10670 ON udata(i,2) GOTO 10680,10700
10680 adr=udata(i,1)-1:ziel=wert:GOSUB 14100
10690 pw(1)=of:GOTO 10720
10700 pw(2)=INT(wert/256)
10710 pw(1)=wert-pw(2)*256
10720 PRINT#sor,"**** Ligne "udata(i,0);" : ";ulata$(i);"=&";HEX$(wert
,4)
10730 FOR k=1 TO udata(i,2)
10740 POKE udata(i,1)+k-1,pw(k)
10750 NEXT k
10760 FOR k=i TO ultp-1
10770 ulata$(k)=ulata$(k+1)
10780 FOR c=0 TO 2:udata(k,c)=udata(k+1,c):NEXT c:NEXT k
10790 ultp=ultp-1:i=i-1
10800 GOTO 10650
10810 REM bef! / instruction 1 octet
10811        sans operande ----------
10820 IF a$(j+1)<>"" THEN 13270
10830 pw(1)=wb1((po-1)/4)
10840 GOTO 13100
```

```
10850 REM ed / 2 octets sans operande
10851 '       debut ed---------------
10860 IF a$(j+1)<>"" THEN 13270
10870 pw(2)=wed((oo-1)/4)
10880 GOTO 13100
10890 REM pseudo instructions ---------
10900 j=j+1
10910 ope$=a$(j):op$=UPPER$(ope$)
10920 ON (oo-1)/4 GOTO 10980,11040,11060,11080,11100,11160
10930 REM EQU
10940 IF label$="" THEN 13280
10950 a$=op$:GOSUB 13790
10960 wlta(ltp-1)=wert
10970 GOTO 13100
10980 REM ORG
10990 IF op$="" THEN 13290
11000 a$=op$:GOSUB 13790
11010 lp=0
11020 mpc=wert:mostart=mpc
11030 GOTO 13100
11040 REM END
11050 GOTO 13400
11060 REM DB
11070 a$=op$:GOSUB 14050:GOTO 13100
11080 REM DW
11090 a$=op$:GOSUB 13860:GOTO 13100
11100 REM DM
11110 IF LEFT$(op$,1)<>CHR$(34) OR RIGHT$(op$,1)<>CHR$(34) THEN 13260
11120 zwi$=MID$(ope$,2,LEN(ope$)-2)
11130 lp=LEN(zwi$)
11140 FOR i=1 TO lp:pw(i)=ASC(MID$(zwi$,i,1)):NEXT
11150 GOTO 13100
11160 REM DS
11170 a$=op$:GOSUB 13860
11180 ds=wert:lp=0:GOTO 13100
11190 REM evaluation instr -----------
11200 j=j+1:ope$=a$(j)
11210 op$=UPPER$(ope$)
```

```
11220 IF op$="" AND bef$<>"RET " THEN 13290
11230 GOSUB 11240:GOTO 11340
11240 poko=INSTR(op$,",")
11250 IF poko=0 THEN o1$=op$:koflag=0:GOTO 11280
11260 koflag=-1
11270 o1$=LEFT$(op$,poko-1):o2$=RIGHT$(op$,LEN(op$)-poko)
11280 pokla=INSTR(op$,"("):poklz=INSTR(op$,")")
11290 IF pokla=0 THEN klaflag=0:klin$="":GOTO 11330
11300 IF pokla>poklz THEN GOTO 13260
11310 klaflag=-1
11320 klin$=MID$(op$,pokla+1,poklz-pokla-1)
11330 RETURN
11340 REM
11350 ipo=INSTR(op$,"IX")
11360 IF ipo<>0 THEN pwi=&DD:ireg$="IX":GOTO 11450
11370 ipo=INSTR(op$,"IY")
11380 IF ipo<>0 THEN pwi=&FD:ireg$="IY":GOTO 11450
11390 zwi=(op+3)/4
11400 ON zwi GOTO 12630,11920,11900,12040,12040,12080,12220,12240,1234
0,12320,12380,12430,12430,12520,12560
11410 REM ld0,sautrelatif(2),saut(3),nombre(2),pile(2),rst,i/o,im
11420 IF zwi<24 THEN 11590
11430 IF zwi<32 THEN 11760
11440 GOTO 11830
11450 REM instructions indexees ------
11460 iflag=-1
11470 IF (NOT klaflag) OR (ipo-pokla<>1) THEN op$=LEFT$(op$,ipo-1)+"HL
"+RIGHT$(op$,LEN(op$)-ipo-1):GOTO 11550
11480 zwi$=MID$(klin$,3,1):IF zwi$<>"+" AND zwi$<>"-" THEN IF bef$="JP
  " THEN 11540 ELSE GOTO 13250
11490 a$=RIGHT$(klin$,LEN(klin$)-3)
11500 dis$=a$:GOSUB 14050:lp=lp-1
11510 IF fe$<>"" THEN GOTO 13260
11520 disflag=-1
11530 disw=wert:IF zwi$="-" THEN disw=(disw XOR 255) +1
11540 op$=LEFT$(op$,pokla)+"HL"+RIGHT$(op$,LEN(op$)-poklz+1)
11550 IF (INSTR(op$,"IX")=0)AND(INSTR(op$,"IY")=0)THEN 11570
11560 IF (op$=("HL,"+ireg$))AND(bef$="ADD ") THEN op$="HL,HL" ELSE GOT
```

```
0 13260
11570 GOSUB 11240
11580 GOTO 11390
11590 REM arilog ----------------------
11600 IF NOT koflag THEN a$=o1$:GOTO 11620
11610 IF o1$<>"A" THEN 11670 ELSE a$=o2$
11620 lp=1:code=zwi-16
11630 GOSUB 13680
11640 IF rflag THEN pw(1)=128 OR(code*8) OR rrr:GOTO 13100
11650 pw(1)=&X11000110 OR (code*8)
11660 GOSUB 14050:GOTO 13100
11670 IF o1$<>"HL" THEN 13260
11680 a$=o2$
11690 GOSUB 13730
11700 IF NOT rflag THEN 13260
11710 IF bef$="ADD " THEN code=&X1001:lp=1:GOTO 11750
11720 pw(1)=&ED:lp=2
11730 IF bef$="ADC " THEN code=&X1001010 :GOTO 11750
11740 IF bef$="SBC " THEN code=&X1000010 ELSE GOTO 13250
11750 pw(lp)=code OR (dd*16):GOTO 13100
11760 REM rotation -------------------
11770 lp=2:pw(1)=&CB
11780 IF koflag THEN 13260
11790 a$=op$:GOSUB 13680
11800 IF NOT rflag THEN 13260
11810 pw(2)=(8*(zwi-24)) OR rrr
11820 GOTO 13100
11830 REM bitti ----------------------
11840 lp=2:pw(1)=&CB
11850 a$=o2$:GOSUB 13680
11860 IF NOT rflag THEN 13260
11870 bbb=ASC(o1$)-48
11880 IF (0>bbb) OR (7<bbb) OR (LEN(o1$)<>1) THEN 13260
11890 pw(2)=(64*(zwi-31))OR(bbb*8)OR rrr:GOTO 13100
11900 REM sauts relatifs -------------
11910 lp=1:pw(1)=&10:a$=op$:GOTO 11990
11920 lp=1
11930 IF NOT koflag THEN ccc=&X11:a$=op$:GOTO 11980
```

```
11940 a$=o1$:GOSUB 13760
11950 IF (NOT cflag) OR (ccc>3) THEN 13260
11960 ccc=ccc OR 4
11970 a$=o2$
11980 pw(1)=ccc*8
11990 IF LEFT$(a$,1)<>"$" THEN GOSUB 13860:lp=lp-2:IF i>ltp THEN wert=
mpc  :GOTO 12010:ELSE 12010
12000 wert=mpc+VAL(RIGHT$(a$,LEN(a$)-1))
12010 lp=lp+1:adr=mpc:ziel=wert
12020 GOSUB 14100
12030 pw(2)=of:GOTO 13100
12040 REM sauts ----------------------
12050 zwi=1:lp=1
12060 IF bef$="RET " THEN code=0 ELSE code=&X100
12070 GOTO 12110
12080 IF op$="(HL)" THEN lp=1:pw(1)=&E9:GOTO 13100
12090 code=&X10
12100 zwi=0:lp=1
12110 IF bef$="RET " THEN IF op$="" THEN 12130 ELSE 12160 ELSE
12120 IF koflag THEN 12160
12130 pw(1)=192 OR code OR 1 OR (zwi*8)
12140 a$=op$
12150 GOTO 12200
12160 a$=o1$:GOSUB 13760
12170 IF NOT cflag THEN 13260
12180 pw(1)=192 OR code OR(ccc*8)
12190 a$=o2$
12200 IF bef$="RET " THEN 13100
12210 GOSUB 13860:GOTO 13100
12220 REM Instructions de comptage ----
12230 zwi=0:GOTO 12250
12240 zwi=1
12250 IF koflag THEN 13260
12260 lp=1:a$=op$:GOSUB 13680
12270 IF rflag THEN pw(1)=&X100 OR (rrr*8) OR zwi:GOTO 13100
12280 GOSUB 13730
12290 IF NOT rflag THEN 13260
12300 pw(1)=&X11 OR (dd*16) OR (zwi*8)
```

```
12310 GOTO 13100
12320 REM Instructions de pile --------
12330 code=&X11000001:GOTO 12350
12340 code=&X11000101
12350 a$=op$:dreg$(3)="AF":GOSUB 13730:dreg$(3)="SP"
12360 IF NOT rflag THEN 13260
12370 lp=1:pw(1)=code OR (dd*16):GOTO 13100
12380 REM restart ---------------------
12390 a$=op$:GOSUB 14050
12400 zwi=wert1/8
12410 IF zwi<>INT(zwi) OR zwi>7 THEN 13260
12420 lp=1:pw(1)=&X11000111 OR (zwi*8):GOTO 13100
12430 REM Instructions I/O ------------
12440 IF NOT(koflag AND klaflag) THEN 13260
12450 IF bef$="IN  " THEN zwi=0 ELSE zwi=1:zwi$=o2$:o2$=o1$:o1$=zwi$
12460 IF klin$<>"C" THEN 12500
12470 a$=o1$:GOSUB 13680
12480 IF (NOT rflag) OR (klin$<>"C") THEN 13260
12490 lp=2:pw(1)=&ED:pw(2)=64 OR (rrr*8) OR zwi:GOTO 13100
12500 lp=1:a$=klin$:GOSUB 14050
12510 pw(1)=&X11011011 XOR (zwi*8):GOTO 13100
12520 REM interrupt modi --------------
12530 IF op$<>"0" AND op$<>"1" AND op$<>"2" THEN 13260
12540 lp=2:pw(1)=&ED
12550 pw(2)=&X1000110 OR ((VAL(op$)-(op$<>"0"))*8):GOTO 13100
12560 REM EX -------------------------
12570 lp=1
12580 IF op$="(SP),HL" THEN pw(1)=&E3:GOTO 13100
12590 IF op$="DE,HL" THEN pw(1)=&EB:GOTO 12620
12600 IF op$="AF,AF'" THEN pw(1)=&8:GOTO 13100
12610 GOTO 13260
12620 IF iflag THEN 13260 ELSE 13100
12630 REM ld---------------------------
12640 IF NOT koflag THEN 13260
12650 a$=o1$:GOSUB 13680
12660 IF rflag THEN 12860
12670 GOSUB 13730
12680 IF rflag THEN 12760
```

```
12690 a$=o2$:GOSUB 13730
12700 IF rflag THEN 12740
12710 zwi$=o2$:o2$=o1$:o1$=zwi$
12720 a=0:GOSUB 12940
12730 IF nflag THEN 13260 ELSE GOTO 13100
12740 IF NOT klaflag THEN 13260
12750 zwi$=o2$:zwiflag=1:GOTO 12800
12760 IF op$="SP,HL" THEN lp=1:pw(1)=&F9:GOTO 13100
12770 IF klaflag THEN zwi$=o1$:zwiflag=0:GOTO 12800
12780 a$=o2$
12790 lp=1:code=1:GOTO 12830
12800 a$=klin$
12810 IF zwi$="HL" THEN lp=1:code=&A:GOTO 12830
12820 lp=2:pw(1)=&ED:code=&X1001011
12830 code=code AND NOT (zwiflag*8)
12840 pw(lp)=code OR (dd*16)
12850 GOSUB 13860:GOTO 13100
12860 zzz=rrr:a$=o2$:GOSUB 13680
12870 IF NOT rflag THEN 12900
12880 lp=1:pw(1)=64 OR (zzz*8) OR rrr
12890 IF pw(1)=&76 THEN 13260 ELSE 13100
12900 a=1:GOSUB 12940
12910 IF NOT nflag THEN 13100
12920 lp=1:pw(1)=&X110 OR (rrr*8)
12930 a$=o2$ : GOSUB 14050:GOTO 13100
12940 REM Chargement 8 bits special ---
12950 nflag=0
12960 IF o1$<>"A" THEN nflag=-1:RETURN
12970 IF klaflag THEN 13030
12980 IF o2$="I"THEN zwi=0:GOTO 13010
12990 IF o2$="R" THEN zwi=1:GOTO 13010
13000 nflag=-1:RETURN
13010 code =&X1000111:lp=2:pw(1)=&ED
13020 pp=(a*2) OR zwi:GOTO 13080
13030 IF klin$="BC" THEN zwi=0:GOTO 13070
13040 IF klin$="DE" THEN zwi=1:GOTO 13070
13050 lp=1:pw(1)=&X110010 OR (a*8)
13060 a$=klin$:GOSUB 13860:RETURN
```

```
13070 code=&X10:lp=1:pp=(zwi*2)OR a
13080 pw(lp)=code OR (8*pp):RETURN
13090 REM
13100 REM Sortie **********
13110 IF iflag THEN 13310
13120 IF fe$<>"" THEN feza=feza+1
13130 IF NOT listflag THEN LOCATE 5,3:PRINT zenr:GOTO 13200
13140 IF fe$<>"" THEN PRINT CHR$(7);:PRINT#sor,fe$,TAB(30);zenr;zeia$:
GOTO 13210
13150 PRINT#sor,HEX$(mpc,4);" ";
13160 FOR i=1 TO lp:PRINT#sor,HEX$(pw(i),2);:NEXT i
13170 PRINT#sor,TAB(14);USING"####";zenr;
13180 PRINT#sor,TAB(20);label$;TAB(27);bef$;TAB(32);ope$;" ";bemer$;
13190 PRINT#sor
13200 FOR i=1 TO lp:POKE mpc+i-1,pw(i):NEXT i
13205 mpc=mpc+lp+ds
13210 lp=0:ds=0
13220 label$="":bef$="":ope$="":bemer$="":fe$=""
13230 GOTO 10130
13240 REM Messages d'erreur -----------
13250 fe$="Syntax Error":GOTO 13100
13260 fe$="Syntax Error dans l'operande":GOTO 13100
13270 fe$="Operande en trop":GOTO 13100
13280 fe$="Label manquant":GOTO 13100
13290 fe$="Operande manquant":GOTO 13100
13300 fe$="illegal Quantity":GOTO 13100
13310 REM indexe ---------------------
13320 FOR j=lp TO 1 STEP -1
13330 pw(j+1)=pw(j):NEXT
13340 pw(1)=pwi:lp=lp+1
13350 IF NOT disflag THEN 13380
13360 IF lp=3 THEN pw(4)=pw(3)
13370 pw(3)=disw:lp=lp+1
13380 iflag=0:disflag=0
13390 GOTO 13120
13400 REM Fin du programme **********
13410 PRINT#sor
13420 IF ultp=0 THEN 13470
```

```
13430 FOR i=0 TO ultp-1
13440 PRINT#sor,"Label non-defini";ulata$(i);" en";udata(i,0);" / Adre
sse &";HEX$(udata(i,1),4)
13450 feza=feza+1:NEXT i
13460 PRINT#sor
13470 PRINT#sor,"Programme :";nom$
13480 PRINT#sor,"Debut : &";HEX$(mpstart,4);"    Fin : &";HEX$(mpc-1,4
)
13490 PRINT#sor,"Longueur : ";HEX$(mpc-mpstart,4)
13500 PRINT#sor,feza;:IF feza<2 THEN PRINT#sor," Erreur" ELSE PRINT#so
r," Erreurs"
13510 IF ltp=0 THEN 13560
13520 PRINT#sor,"Table de variables :"
13530 FOR i=0 TO ltp-1
13540 PRINT#sor,LEFT$(lata$(i)+"        ",7);HEX$(wlta(i),4),
13550 NEXT i
13560 PRINT#sor
13570 INPUT"Sauvegarde (o/n):",t$
13580 IF t$<>"o" THEN 13600
13590 SAVE nom$+".obj",B,mpstart,mpc-mpstart
13600 END
13610 REM Sous-programmes **********
13620 REM test label ------------------
13630 laas=ASC(UPPER$(LEFT$(a$,1)))
13640 IF laas<65 OR laas>90 THEN nolafl=-1:RETURN
13650 IF LEN(a$)>6 THEN PRINT"Label trop long":a$=LEFT$(a$,6)
13660 lab$=a$:nolafl=0
13670 RETURN
13680 REM test r ----------------------
13690 FOR i=0 TO 7
13700 IF reg$(i)=a$ THEN rflag=-1:rrr=i:RETURN
13710 NEXT
13720 rflag=0:RETURN
13730 REM test rps --------------------
13740 FOR i=0 TO 3:IF dreg$(i)=a$ THEN rflag=-1:dd=i:RETURN ELSE NEXT
13750 rflag=0:RETURN
13760 REM test cond -------------------
13770 FOR i=0 TO 7:IF a$=cond$(i) THEN cflag=-1:ccc=i:RETURN ELSE NEXT
```

```
13780 cflag=0:RETURN
13790 REM test nombre -----------------
13800 wert=VAL(a$)
13810 laas=ASC(LEFT$(a$,1))
13820 IF wert=0 AND laas<>38 AND (laas>57 OR laas<48) THEN fe$="illega
l character":wert=0:RETURN
13830 IF wert>=0 THEN 13850
13840 IF LEFT$(a$,1)="&" THEN wert=wert+2^16 ELSE fe$="illegal Quantit
y":wert=0
13850 RETURN
13860 REM Evaluateur ------------------
13870 GOSUB 13620
13880 IF nolafl THEN GOSUB 13790:GOTO 13920
13890 FOR i=0 TO ltp:IF lata$(i)<>lab$ THEN NEXT
13900 IF i>ltp THEN 13980
13910 wert=wlta(i)
13920 werth=INT(wert/256)
13930 wertl=wert-256*werth
13940 lp=lp+2
13950 pw(lp-1)=wertl
13960 pw(lp)=werth
13970 RETURN
13980 ulata$(ultp)=a$
13990 udata(ultp,0)=zenr
14000 udata(ultp,1)=mpc+lp-iflag-disflag
14010 udata(ultp,2)=2+(bef$="DJNZ" OR bef$="JR  ")
14020 ultp=ultp+1
14030 wert=0
14040 GOTO 13920
14050 REM Evaluateur low --------------
14060 GOSUB 13860
14070 lp=lp-1
14080 IF werth<>0 THEN fe$="illegal Quantity":wert=0
14090 RETURN
14100 REM Calculer offset -------------
14110 of =ziel-adr
14120 of=of-2
```

```
14130 IF of>129 OR of<-126 THEN fe$="illegal Quantity":of=0
14140 IF of<0 THEN of=256+of
14150 RETURN
14160 REM Initialisation ***********
14170 zei$="test"
14180 vapt=HIMEM-FRE(0)-15
14190 DEF FNdeek(x)=PEEK(x)+256*PEEK(x+1)
14200 MODE 2
14210 teadr$="LD  JR  DJNZCALLRET JP  INC DEC PUSHPOP RST IN  OUT IM
EX  ADD ADC SUB SBC AND XOR OR  CP  RLC RRC RL  RR  SLA SRA **
* SRL BIT RES SET "
14220 teed$="CPD CPDRCPI CPIRIND INDRINI INIRLDD LDDRLDI LDIRNEG OTDRO
TIROUTDOUTIRETIRETNRLD RRD "
14230 DATA A9,B9,A1,B1,AA,BA,A2,B2,A8,B8,A0,B0,44,BB,B3,AB,A3,4D,45,6F
,67
14240 tebl$="CCF CPL DAA DI  EI  EXX HALTNOP RLA RLCARRA RRCASCF "
14250 DATA 3F,2F,27,F3,FB,D9,76,00,17,07,1F,0F,37
14260 teps$="EQU ORG END DB  DW  DM  DS  "
14270 DIM lata$(50),wlta(50),ulata$(50),udata$(50,2)
14280 DIM wbl(12),wed(20)
14290 FOR i=0 TO 20:READ a$:wed(i)=VAL("&"+a$):NEXT
14300 FOR i=0 TO 12:READ a$:wbl(i)=VAL("&"+a$):NEXT
14310 opc=384:mpc=40960:mpstart=mpc
14320 DIM reg$(7),cond$(7),dreg$(3)
14330 FOR i=0 TO 7:READ reg$(i):NEXT
14340 FOR i=0 TO 7:READ cond$(i):NEXT
14350 FOR i=0 TO 3:READ dreg$(i):NEXT
14360 DATA B,C,D,E,H,L,(HL),A
14370 DATA NZ,Z,NC,C,PO,PE,P,M
14380 DATA BC,DE,HL,SP
14390 GOTO 10020
```

Description du programme :

Ligne 1 :

La zone mémoire RAM &A000-&AB7F est réservée pour le programme machine, puis le programme source qui se trouve entre les lignes 2 et 9999 est sauté.

Ligne 10010 :

Saut à la section de programme initialisation, c'est-à-dire mise en place de la table d'instructions, etc... (voir ligne 14160-).

Lignes 10020-10090 :

Le menu, le flag list (V) et le canal de sortie sont fixés.

Lignes 10100-10190 :

bpc indique l'adresse actuelle dans le programme source (bpc=Basic Programm Counter). Au début d'une ligne figure la longueur de celle-ci en format octet faible-octet fort. FNdeek(bpc) lit la valeur 16 bits à l'adresse bpc et bpc+1. Cette valeur correspond à la longueur de la ligne laze (V). bpc est incrémenté de 2 et le numéro de ligne zenr (V) est lu. S'il est supérieur à 9999, l'assemblage est terminé. La ligne 10180 teste si le caractère ' figure bien en début de ligne. S'il ne s'y trouve pas, le message d'erreur est sorti et on lit la prochaine ligne.

Lignes 10200-10240 :

Cette section de programme affecte à zei$ (V) la ligne actuelle. Pour que l'assembleur tourne aussi vite que possible, ceci est réalisé par une modification des pointeurs de chaîne de zei$ (V) dans la table interne des variables.

Lignes 10240-10420 :

Un commentaire éventuel est d'abord stocké dans bemer$ (V) puis le reste de la ligne est découpé en autant de morceaux qu'il y a de caractères espace et les différents morceaux sont stockés dans a$(j) (V). Si la ligne a été découpée en plus de trois morceaux (label, instruction, opérande), donc si j>2, un Syntax error est sorti.

Lignes 10430-10540 :

On teste ici si a$(j) (V) constitue bien une instruction valable. Si l'instruction est valable, on saute à l'endroit où ces instructions sont assemblées.

Lignes 10540-10550 :

Si aucune instruction n'a été identifiée, on teste s'il s'agit d'une pseudo-instruction et on saute aux pseudo-instructions.

Lignes 10560-10800 :

S'il s'agit d'un label, il est entré dans la table des labels et la valeur du mpc (Machine Programm Counter) lui est affectée (lignes 10610-10630). Dans les lignes suivantes jusqu'à la ligne 10800 on teste si ce label a déjà été utilisé alors qu'il n'était pas encore défini. Si c'était le cas, la valeur correspondante est POKEe et le label est effacé de la table des non-définis ulata$(i) (V). S'il ne s'agit pas d'un label valable (parce qu'il ne commence pas par une lettre), le message d'erreur "label manquant" est sorti (ligne 10600).

Lignes 10810-10840 :

Ici sont évaluées les instructions avec un code d'opération sur un octet, qui ne possèdent pas d'opérande. Le code est déterminé à partir de l'emplacement dans teb1$ (V) et du wb1(i) correspondant (V).

Lignes 10850-10880 :

Ici sont traités les instructions sur deux octets sans opérande. Le premier code d'opération est toujours &ED (pw(1)=&ED). Le deuxième octet du code d'opération est déterminé à partir de l'emplacement de l'instruction dans teed$ (V) et de wed(i) (V).

Lignes 10890-13080 :

Ici sont traduites les pseudo-instructions.

Lignes 11190-13080 :

Si l'instruction ne relève d'aucun des groupes sus-nommés, elle est évaluée dans cette section du programme. On teste d'abord si l'opérande op$ (V) contient une virgule. Si oui, il est décomposé en o1$ (V) (partie avant la virgule) et o2$ (V) (partie après la virgule) et koflag est fixé à -1 (=vrai). On teste ensuite si des parenthèses apparaissent. Si oui, le contenu des parenthèses est stocké dans klin$ (V) et klflag (V) est fixé sur -1.

Lignes 11280-11330 :

S'il s'agit d'une instruction avec adressage indexé, on saute à la ligne 11450.

Lignes 11400-11440 :

On saute ici à la routine de traitement des instructions qui convient en fonction de la position dans teadr$ (V).

Lignes 11450-11580 :

Pour les instructions indexées, XY ou XY+dis sont remplacés par HL, le iflag et éventuellement le disflag (V) sont mis. On poursuit ensuite l'évaluation normale de l'instruction en ligne 11390.

Après l'interprétation, le remplacement effectué est annulé et le code de l'instruction indexée (qui est analogue à celui de HL) est sorti.

Lignes 11590-11750 :

Les instructions arithmétiques (8 et 16 bits) sont interprétées ici.

Lignes 11760-11820 :

Instructions de rotation et de décalage.

Lignes 11830-11890 :

Instructions de manipulation de bits.

Lignes 11900-12030 :

Sauts relatifs (JR et DJNZ).

Lignes 12050-12210 :

Autres sauts (JP, RET, CALL).

Lignes 12220-12310 :

Instructions de comptage (INC, DEC).

Lignes 12320-12620 :

(voir lignes REM)

Lignes 12630-13080 :

Instructions de chargement.

Il ne nous est pas possible dans le cadre de cet ouvrage d'expliquer chaque routine. Nous allons donc expliquer à titre d'exemple la routine pour les instructions de manipulation de bits :

Ligne 11840 :

lp (V) (longueur de l'instruction, c'est-à-dire nombre de valeurs à POKEr) vaut 2. La première valeur pw(1) (V) est &CB. Il en va de même pour toutes les instructions bits.

Ligne 11850 :

o2$ (V) (partie de l'opérande située après la virgule) est stocké dans a$ (V) pour être transmis au sous-programme commençant en ligne 13680. Le sous-programme détermine s'il s'agit d'un des registres A, B, C, D, E, H, L ou (HL).

Ligne 11860 :

Si aucune identité n'a pu être constatée (rflag=0), le message d'erreur "Syntax error dans l'operande" est sorti. Sinon le code du registre en rrr est renvoyé et le flag rflag est mis.

Ligne 11870 :

bbb reçoit la valeur du nombre (le numéro de bit) figurant devant la virgule.

Ligne 11880 :

On teste maintenant si le nombre est bien compris entre 0 et 7. S'il n'est pas dans ces limites, le message "Syntax error dans l'operande" est sorti.

Ligne 11890 :

Pour finir, le code d'opération est stocké dans pw(2). Le code d'opération est ainsi constitué :

```
01 bbb rrr - pour les instructions BIT
10 bbb rrr - pour les instructions RES
11 bbb rrr - pour les instructions SET
```

(zwi-31)*64 donne les bits 7 et 6 du code d'opération. zwi (V) est l'emplacement de l'instruction dans teadr$ (V). bbb*8 représente les bits 5-3 et rrr les bits 2-0. rrr a été déterminé par le sous-programme et correspond au code du registre. Une fois le code d'opération calculé, on saute à la sortie (ligne 13100). Les autres routines fonctionnent de façon semblable.

Lignes 13100-13230 :

Sortie : si le iflag (V) (flag pour les instructions indexées) est mis, on saute d'abord à la routine spéciale en ligne 13310. Sinon la ligne assembleur complète est sortie ici. Si des erreurs se sont produites, celles-ci sont indiquées et feza (V), le compteur d'erreur est augmenté.

Avant de sauter au début, pour assembler la ligne suivante, les variables importantes sont annulées et mpc (Machine Programm Counter) est augmenté de la longueur de l'instruction lp (V).

Lignes 13240-13300 :

Si une erreur a été constatée, on saute à l'une de ces routines qui affecte à fe$ (V) la chaîne d'erreur et saute ensuite à la sortie du message.

Lignes 13310-13390 :

Ici sont préparés les codes pour les instructions indexées.

Lignes 13400-13600 :

A la fin du programme, les labels non-définis sont sortis ainsi que le nom du programme, les adresses de début et de fin, la longueur, le nombre d'erreur et la table des variables.

En ligne 13570 est réalisée la sauvegarde du code objet.

En lignes 13610-14150 figurent
des sous-programmes souvent utilisés :

Lignes 13620-13670 :

On teste ici si a$ (V) contient un label correct.

Lignes 13680-13770 :

Ces lignes testent si a$ (V) contient un registre (A, B, C, D, E, H, L, (HL)).

Lignes 13730-13750 :

On teste ici si a$ (V) contient une des paires de registres BC, DE, HL, SP.

Lignes 13760-13780 :

Teste si a$ contient une condition: C, NC, Z, NZ, PO, PE, P, M.

Lignes 13790-13850 :

Teste si a$ (V) est un nombre et renvoie la valeur de ce nombre.

Lignes 13860-14040 :

Ces lignes déterminent la valeur sur deux octets de a$ (V). a$ peut être un nombre, une variable ou un label.

Lignes 14050-14090 :

Détermine la valeur sur un octet (octet faible) de a$.

Lignes 14100-14150 :

Calcule l'offset pour les sauts relatifs.

Lignes 14160-14390 :

Initialisation : les champs de données et les chaînes sont produites pour les comparaisons. vapt (V) indique l'adresse où est stockée la longueur de chaîne de zei$ (V).

FNdeek(X) fournit la valeur 16 bits de deux cases mémoire consécutives.

bpc est fixé sur le début de la ligne suivant la ligne 1 (=384).

mpc est fixé sur &A000.

Liste de variables :
(SUB signifie : sous-programme)

a	transmis au SUB "Chargement spécial 8 bits"
a$	transmis à divers sous-programmes
adr	transmis au SUB "calculer offset": adresse d'origine du saut
aus	canal du périphérique de sortie (0 ou 8)
bbb	code numéro de bit pour instructions de manipulation de bits
bef$	mot d'instruction assembleur
bemer$	commentaire de la ligne assembleur
bepo	position du commentaire dans une ligne donnée
bpc	pointeur de programme Basic
ccc	code de condition pour les sauts
cflag	mis (c'est-à-dire =-1) si condition trouvée annulé (=0) sinon, renvoyé par le SUB "cond test"
code	utilisé pour produire le code d'opération des différentes instructions
dis$	contient la distance des instructions indexées
disflag	mis pour une instruction indexée avec indication de distance, annulé sinon
disw	valeur de la distance indiquée (en complément à 2)
ds	contient le nombre de cases mémoire réservées par une instruction DS
fe$	message d'erreur

feza	nombre d'erreurs
i, j, k	compteurs pour les boucles
iflag	mis pour les instructions indexées, sinon annulé
ipo	position du registre d'index (IX ou IY) dans l'opérande
ireg$	s'il y a adresse indexée, ireg$ contient IX ou IY
klaflag	mis s'il y a des parenthèses dans l'opérande, sinon annulé
klin$	contient le contenu entre parenthèses dans l'opérande (s'il y a lieu)
koflag	mis s'il y a une virgule dans l'opérande, sinon annulé
laas	code ASCII du premier caractère d'un label à tester (SUB "test label")
lab$	nom de label renvoyé par le SUB test label
label$	nom actuel de label
laze	longueur de la ligne de programme source qui est en train d'être assemblée
listflag	mis si listing souhaité, sinon annulé
lp	longueur d'une instruction (longueur du code objet)
ltp	pointeur sur la place libre dans la table des labels (lata$) (Pointeur dans Table des Labels)
mpc	Machine Programme Counter: indique la case mémoire dans laquelle sera stocké le prochain code machine
mpstart	adresse de début du programme machine
nom$	nom de programme
nflag	mis s'il y a adressage immédiat pour une instruction de chargement, sinon annulé (SUB"chargement special 8 bits)
nolafl	NO LAbel FLag: mis lorsque le test label a été négatif, sinon annulé. Renvoyé par le SUB test label.
o1$	partie avant la virgule de l'opérande
o2$	partie après la virgule de l'opérande
of	offset calculé par le SUB offset
op$	opérande pour le traitement
ope$	opérande d'origine pour la sortie
po	position du mot instruction dans la chaîne test
pokla	position de la parenthèse ouverte dans l'opérande
poklz	position de la parenthèse fermée dans l'opérande
poko	position de la virgule dans l'opérande
pwi	premier octet code d'opération pour l'adressage indexé, soit &FF ou &DF
rflag	mis si le SUB "rtest" a identifié l'un des registres A, B, C, D, E, H, L, (HL), sinon annulé.

rrr	code du registre; renvoyé par le SUB "rtest"
sppo	SPace POsition, position du caractère espace dans la ligne
t$	chaîne d'entrée (menu)
teadr$	TEst ADRessage: contient tous les mots d'instruction qui se présentent avec un opérande
teb1$	TEst 1 Byte: contient tous les mots d'instruction qui n'ont jamais d'opérande et qui ont un code d'opération sur un octet
teed$	TEst &ED: contient tous les mots d'instruction qui n'ont jamais d'opérande et qui ont un code d'opération sur deux octets dont le premier octet est &ED
teps$	TEst PSeudo: contient toutes les pseudo-instructions
ultp	Undefined Label Table Pointer (pointeur dans la table des labels non-définis): pointe sur le prochain emplacement libre dans les tables ulata$ ou udata
vapt	VAriable PoinTer (pointeur de variable): indique l'adresse de zei$ dans la table interne des variables
wert	valeur d'une expression, renvoyée par le SUB "Evaluateur" ou le SUB "Test nombre"
werth	octet fort de wert
wertl	octet faible de wert
zei$	contient la ligne actuellement traitée
zeia$	contient la ligne actuelle (sous sa forme originelle)
zenr	numéro de ligne actuel
ziel	transmis par le SUB "calculer offset" à l'adresse objet
zwi	diverses tâches de stockage intermédiaire
zwi$	diverses tâches de stockage intermédiaire

Tables

lata$(50)	table des labels
wlta(50)	valeurs des labels de la table des valeurs
ulata$(50)	table des labels non-définis
udata$(50,2)	données se rapportant aux labels non-définis
(i,0)	numéro de ligne où apparaît ...
(i,1)	adresse de la valeur à POKEr ensuite
(i,2)	type, c'est-à-dire 16 bits (=2) ou offset (=1)
wb1(12)	codes d'opération des instructions sur un octet (teb1$)

wed$(20)	codes d'opération des instructions sur deux octets (teed$)
reg$(7)	table des registres: B, C, D, E, H, L, (HL), A
cond$(7)	table des conditions: NZ, Z, NC, C, PO, PE, a, M
dreg$(3)	table des doubles registres: BC, DE, HL, SP

5.2. PROGRAMMATION

Pour notre premier projet de programme d'envergure, nous allons une fois de plus nous intéresser à l'écran.

Il vous est peut-être déjà arrivé, lors de la programmation de nos programmes d'exemple, de lancer ces programmes sans avoir entré auparavant MODE 2. Vous avez alors été certainement très surpris du résultat produit. Nous allons vous expliquer maintenant ce phénomène :

Après que vous ayez entré MODE 2, la première case de l'écran en haut à gauche correspond à l'adresse &C000 :

>POKE &C000,255< vous permet d'obtenir un trait dans cet emplacement.

Amenez maintenant le curseur sur le bord inférieur de l'écran et faites glisser l'écran une fois (scrolling vers le haut) en appuyant une fois sur la touche curseur bas. Amenez ensuite le curseur au début de la ligne du milieu de l'écran. Si vous entrez à nouveau >POKE &C000,255<, le trait apparaît dans le bas de l'écran. Si vous entrez par contre >POKE &C050,255<, le trait apparaît à nouveau dans l'ancien emplacement. La différence entre &C000 et &C050 est &50, soit 80 en décimal ce qui correspond au nombre de caractères de la ligne qui sont été "expulsés" de l'écran lors du scrolling. Si vous faites à nouveau glisser l'écran, vous ne pourrez plus obtenir le trait dans cette même position qu'avec >POKE &C0A0,255< (&C050+&50=&C0A0).

La différence entre &C000 et l'adresse réelle de la case supérieure gauche de l'écran est stocké de façon interne aux adresses &B1C9

(octet faible) et &B1CA (octet fort). Lisons la valeur 16 bits de ces cases mémoire. S'il n'y a pas eu entre temps d'autre scrolling, nous obtenons la valeur :

Attention : Les adresses &B1C9 et &B1CA sont valables pour le CPC 464. Si vous possédez un CPC 664 ou 6128, vous devrez par la suite les remplacer par &B7C4 et &B7C5.

>PRINT HEX$(PEEK(&B1C9)+PEEK(&B1CA)*256)< soit A0.

Cela correspond exactement à la différence entre &C000 et &C0A0. En modifiant le contenu de &B1C9 et &B1CA vous pouvez obtenir dans certains cas des effets intéressants sur l'écran.

Pour toutes les opérations qui concernent l'écran, il faut prendre en compte cette différence.

Nous allons maintenant modifier le programme d'inversion du caractère supérieur gauche en prenant en compte la différence de scrolling.

Il faut d'abord charger &C000 dans HL. Comme nous travaillons avec l'assembleur, nous stockerons &C000 dans une variable. Nous ferons démarrer le programme comme d'habitude à partir de l'adresse &A000. Les premières lignes se présentent ainsi :

```
10 'Bildad EQU &C000 ;adresse de base de l'écran
20 'ORG    &A000
30 'LD HL,Bildad
```

La différence qui convient doit maintenant être ajoutée à l'adresse de base.

```
40 'LD DE,(&B1C9) ;"différence de scrolling"
50 'ADD HL,DE ;calculer adresse de départ
```

L'instruction de chargement sur 16 bits LD DE,(&B1C9) en ligne 50 charge les octets faible et fort dans la paire de registres DE.

Nous procédons ensuite de la même façon que dans le programme du chapitre 4.10.

```
60 'LD DE,&800 ;différence
70 'LD B,8 ;compteur de boucle
80 'encore LD A,(HL) ;matrice de bits actuelle
90 'CPL ;inverser
100 'LD (HL),A ;stocker à nouveau
110 'ADD HL,DE ;additionner différence
```

ADD HL,DE peut cependant aussi avoir pour effet que HL devienne plus grand que &FFFF.

Exemple :

HL=&F9A0DE=&800

Après ADD HL,DE :

HL=&01A0Carry=1

Il est évident qu'il ne peut s'agir de la bonne adresse en mémoire écran puisque c'est nos programmes Basic qui figurent à cette adresse. Les points suivant les points stockés à l'adresse &FFFF figurent à l'adresse &C000.

Si donc une retenue s'est produite (CF=1), il nous faut ajouter &C000 à HL.

Essayez de compléter ce programme en langage-machine.

Solution :

```
120 'CALL C,DIFADD ;sous-programme de correction
130 'DJNZ encore ; répéter 8 fois
140 'RET ;fin du programme principal
150 'DIFADD PUSH DE ;début sous-pro/sauver DE
160 'LD DE,Bildad ;Bildad=&C000
170 'ADD HL,DE ;ajouter à HL
180 'POP DE ;restaurer DE
```

```
190 'RET ;fin du sous-programme
200 'END
```

Explication :

Ligne 120 :

Si une retenue s'est produite, on saute à la routine de correction.

Lignes 150 et 190 :

Toutes les paires de registres ont déjà été utilisées. L'addition 16 bits n'est cependant possible qu'en adressage implicite. C'est pourquoi le contenu de DE est brièvement stocké provisoirement sur la pile pour en être retiré avec POP DE après l'addition.

Assemblez ce programme. Voici le listing assembleur :

```
A000            10   Bildad   EQU  &C000 ;adresse de base de l'écran
A000            20            ORG  &A000
A000 2100C0     30            LD   HL,Bildad
A003 ED5BC9B1   40            LD   DE,(&B1C9) ;"différence de
                                   scrolling"
A007 19         50            ADD  HL,DE ;calculer adresse de départ
A008 110008     60            LD   DE,&800 ;différence
A00B 0608       70            LD   B,8 ;compteur de boucle
A00D 7E         80   ENCORE   LD   A,(HL) ;matrice de bits actuelle
A00E 2F         90            CPL  ;inverser
A00F 77         100           LD   (HL),A ;stocker à nouveau
A010 19         110           ADD  HL,DE ;additionner différence
A011 DC0000     120           CALL C,DIFADD ;sous-programme de
                                   correction
A014 10F7       130           DJNZ encore ; répéter 8 fois
A016 C9         140  RET ;Fin du programme principal
**** Ligne 120 : DIFADD=A017
A017 D5         150  DIFADD   PUSHDE ;Debut sous-prog/sauverDE
A018 1100C0     160           LD   DE,Bildad ;Bildad=&C000
```

```
A01B 19         170         ADD  HL,DE ;ajouter à HL
A01C D1         180         POP  DE ;Restaurer DE
A01D C9         190         RET ;Fin du sous-programme

Programme:Invers
Debut: &A000  Fin : &A01D
Longueur: 001E
  0 Erreur
Table de variables:
BILDAD C000  ENCORE A00D  DIFADD A017
```

Il y a en ligne 130 un saut au label DIFADD. DIFADD ne réapparaît cependant qu'en ligne 150. C'est pourquoi DC0000 est d'abord stocké comme code. Lors de l'assemblage de la ligne 150, l'assembleur rencontre le label DIFADD et indique que ce label a été rencontré en ligne 120. Le code DC0000 est alors automatiquement fixé correctement. C'est également possible de la même façon pour JR et JP. Ce problème se présente lorsqu'un saut en avant figure dans le programme.

Il est nécessaire de traiter de cette manière les sauts en avant car il s'agit d'un assembleur à 1 passage. Cela signifie que l'assembleur ne parcourt le programme source qu'une fois.

Un assembleur à deux passages par contre ne recherche lors du premier parcours que les variables et labels pour leur affecter une valeur. Ce n'est que lors du second passage que commence l'assemblage. Les assembleurs professionnels effectuent plusieurs passages (PASSES). Le 1-pass-assembler est plus judicieux pour nos besoins car il est environ deux fois plus rapide qu'un 2-pass-assembler.

Mais revenons au programme :

Il y a bien sûr encore d'autres solutions de programmes pour résoudre cette tâche. Ce qui est d'abord décisif c'est que le programme résolve le problème posé. Il est cependant judicieux de chercher à obtenir la version la plus courte et la plus rapide possible.

Pour les programmes suivants, nous accorderons moins d'importance à la rapidité d'exécution et à la place mémoire occupée qu'à la clarté des programmes.

HL ne doit jamais être inférieur à &C000. H peut donc avoir les valeurs &C0 à &FF. Pour toutes les valeurs possibles, les deux bits supérieurs (numéros 6 et 7) sont mis. Pour éviter toute erreur, nous pouvons mettre ces deux bits à 1 lors de chaque parcours de la boucle. Le sous-programme en ligne 160 disparaît donc et nous obtenons pour la ligne 130 :

```
130 'SET 6,H
135 'SET 7,H
```

Avec l'opération logique OR, cette tâche peut être résolue encore plus vite (OR permet de mettre des bits à 1).

```
85 'LD C,&X11000000
130 'LD A,H
133 'OR C
135 'LD H,A
```

Les autres programmes de manipulation de l'écran écrits jusqu'à présent peuvent être utilisés de façon universelle en prenant en compte la différence de scrolling. Nous vous laissons le soin d'apporter à ces programmes les modifications nécessaires.

Routine Moniteur Basic

Nous connaissons maintenant le mode de fonctionnement de l'assembleur. D'autres moyens permettent de faciliter le travail en langage-machine. C'est le cas par exemple de ce qu'on appelle un moniteur de langage-machine.

Nous allons ainsi faire d'une pierre deux coups puisque vous allez découvrir des techniques de programmation fondamentales et que vous aurez en résultat un programme de moniteur.

Comme nous l'avons déjà indiqué, la tâche principale d'un programme de moniteur consiste à afficher le contenu de la mémoire. Ceci peut être réalisé en Basic avec PEEK.

Ecrivez un programme qui sorte, après que vous ayez entré l'adresse de départ (V) et l'adresse finale (V), le contenu des cases mémoire situées entre ces deux adresses. Utilisez pour la sortie le format habituel d'un HEX-DUMP (sortie des cases mémoire en format hexadécimal), soit :

```
Hex-Dump de l'adresse &10 à &27:

0010 C3 16 BA C3 10 BA D5 C9 C.:C.:UI
0018 C3 BF B9 C3 B1 B9 E9 00 C?9C19i.
0020 C3 CB BA C3 B9 B9 00 00 CK:C99..
```

Votre programme doit produire une image a peu de choses près semblable à la nôtre.

Veillez à ce que le HEX-DUMP proprement dit soit suivi à droite de la représentation ASCII des codes. Les codes supérieurs à 127 sont auparavant diminués de 128. Les codes qui ne peuvent être représentés (0-31) sont marqués par un point.

Solution :

```
10 REM routine moniteur Basic
20 MODE 1
30 INPUT debut
40 INPUT fin
50 FOR i=debut TO fin STEP 8
60 ascii$=""
70 PRINT HEX$(i,4);" ";
80 FOR j=0 TO 7
90 w=PEEK(i+j)
100 PRINT HEX$(w,2);" ";
110 IF w>127 THEN w=w-128
120 IF w<32 THEN w=46
130 ascii$=ascii$+CHR$(w)
140 NEXT j
```

```
150 PRINT" ";ascii$;
160 NEXT i
170 END
```

Ce programme vous permet maintenant d'examiner la totalité de la RAM de l'ordinateur. Entrez dans votre programme de moniteur la ligne suivante :

```
1 REM ceci est la première ligne
```

Examinons maintenant le contenu de la mémoire de &170 à &200. Dans la représentation ASCII du contenu de la mémoire, vous reconnaissez la première ligne, c'est-à-dire le commentaire "ceci est la première ligne". C'est en effet à partir de &170 que sont stockés dans la mémoire les programmes Basic. Le programme Basic est immédiatement suivi d'une page, gérée de façon interne, qui contient toutes les variables numériques, la valeur du nombre est stockée directement alors que pour les variables alphanumériques (chaînes), ce sont l'adresse et la longueur de la chaîne qui sont stockées. Les variables sont placées dans cette page, dans l'ordre dans lequel elles apparaissent dans le programme.

Les programmes de moniteur professionnels offrent la possibilité de modifier le contenu de la mémoire en modifiant directement l'affichage de ce contenu.

Routine fill

Intéressons-nous maintenant à la routine "fill". Elle est utilisée pour remplir une zone quelconque de la mémoire avec une valeur déterminée. Il est par exemple possible de vider intégralement la mémoire écran en la remplissant à zéros. L'instruction F(-ill) est par exemple utilisée pour réaliser certaines conditions avant l'exécution du programme. Le problème de programmation suivant se pose :

Le programme Basic demande les adresses de début et de fin de la zone à remplir et la valeur avec laquelle cette zone doit être remplie. Le programme Basic doit vérifier si l'adresse de début (V) est bien inférieure à l'adresse finale (V) est bien comprise entre 0 et 255. Ces trois valeurs (soit 5 octets) sont alors "POKEes" dans des cases mémoire définies une fois pour toutes, de sorte qu'après que l'appel de la routine fill en langage-machine, elles soient à la disposition de cette routine. Le programme en langage-machine exécute le remplissage proprement dit qui est suivi d'un retour au Basic.

Voici maintenant un programme Basic qui traite une entrée sous cette forme et qui effectue les tests sur les critères indiqués ci-dessus.

```
10 MEMORY &9FFF
90 MODE 2
100 LOCATE 10,5:PRINT"PROGRAMME DE MONITEUR"
110 LOCATE 5,8:PRINT "ENTER
120 LOCATE 7,10:INPUT"ADRESSE DE DEPART:",DEBUT
130 IF DEBUT<O OR DEBUT>=2^16 THEN 120
140 IF DEBUT<> INT(DEBUT)THEN 120
150 LOCATE 7,11:INPUT"ADRESSE DE FIN:",FIN
160 IF FIN<=DEBUT OR FIN<=2^16 THEN 150
170 IF FIN<>INT(FIN)THEN 150
180 LOCATE 7,12:INPUT"VALEUR:",VALEUR
190IFVALEUR<OORVALEUR>255OR(VALEUR<>INT(VALEUR))THEN180
200 POKE &A000,VALEUR
210 POKE &A002,INT(DEBUT/256):POKE &A001,DEBUT-INT
(DEBUT/256)*256
220 POKE &A004,INT(FIN/256):POKE &A003,FIN-INT(FIN/256)*256
230 CALL &A005
240 END
```

Pour le programme en langage-machine, la valeur (V) se trouve à l'adresse &A000, l'adresse de début se trouve en &A001 (low/high) et l'adresse finale en &A003 (low/high).

Comme les premières cases mémoire à partir de &A000 sont occupées, nous lançons le programme en langage-machine en &A005.

Voici la première partie du programme source :

```
10 'ORG &A005
20 'DEBUT EQU &A001
30 'FIN EQU &A003
40 'VALEUR EQU &A000
50 'LD A,(VALEUR)
60 'LD DE,(DEBUT) ;POINTEUR DE BLOC
```

Description du programme :

Ligne 10 :

Programme de début sur &A005

Lignes 20-40 :

Pour plus de clarté, les adresses des données transmises (adresse de transmission) sont définies comme variables. Pour modifier les adresses de transmission, il suffit de modifier la valeur utilisée dans la définition des variables.

Lignes 50-60 :

La valeur est chargée dans l'accumulateur (1 octet), l'adresse finale dans la paire de registres HL (2 octets) et l'adresse de début dans le registre DE.

Nous en arrivons maintenant à la routine Fill proprement dite. Voici tout d'abord la solution la plus simple :

```
70 'boucle ld (de),a ;ecrire valeur
80 'inc de ;augmenter pointeur
90 'ld hl,(fin) ;calculer
100 'sbc hl,de ;si deja
110 'jr nz,boucle ;fin atteinte?
120 'ld (de),a ;remplir dernier element
130 'ret
140 'end
```

Description du programme :

Ligne 50 :

Charger l'adresse finale (V) dans HL.

Ligne 70 :

Début de la boucle. La valeur (A) est stockée à l'adresse HL.

Ligne 80 :

Le pointeur d'adresse (DE) est augmenté.

Ligne 100 :

Soustraction sur 16 bits : adresse actuelle ôtée de l'adresse finale (HL-DE).

Ligne 110 :

Si le pointeur d'adresse DE est plus petit que l'adresse finale en HL, le flag Z n'est pas mis puisque HL-DE est différent de 0. Dans ce cas, (NZ), on saute au début de la boucle (boucle). Si HL est par contre égal à DE, Z=1 et l'instruction suivante (ligne 120) est exécutée.

Ligne 120 :

La valeur A (=contenu de l'accumulateur) est également écrite dans l'adresse finale de la zone à remplir. Cela n'avait pas encore été fait ! ! (pourquoi ?).

Ligne 130 :

Retour au Basic

Si vous faites assembler ce programme par l'assembleur, vous obtenez le listing assembleur suivant :

```
A005                10           ORG &A005
A005                20 DEBUT     EQU &A001
A005                30 FIN       EQU &A003
A005                40 VALEUR    EQU &A000
A005   3A0AO        50           LD  A,(VALEUR)
A008   ED5B01A0     60           LD  DE,(DEBUT) ;POINTEUR DE BLOC
A00C   12           70 BOUCLE    LD  (de),a ;ecrire valeur
A00D   13           80           INC de ;augmenter pointeur
A00E   2A03A0       90           LD  hl,(fin) ;calculer
A011   ED52        100           SBC hl,de ;si deja
A013   20F7        110           JR  nz,boucle ;fin atteinte?
A015   12          120           LD  (de),a ;remplir dernier element
A016   C9          130           RET

Programme : Fill
Debut : &A005        Fin &A016
Longueur : 0012
  0  Erreur
Table de variables :
DEBUT  A001  FIN  A003  VALEUR  A000  BOUCLE  A00C
```

Ecrivez un programme de chargement Basic pour ce programme et intégrez-le dans le programme Basic Fill.

```
20 FOR I=&A000 TO &A016:READ a$:a=VAL("&"+a$):POKE i,a:NEXT
25 DATA ff,00,c0,ff,ff
30 DATA 3a,00,a0,ed,5b,01,a0,12
40 DATA 13,2a,03,a0,ed,52,20,f7
50 DATA 12,c9
```

Comme nous l'avons déjà indiqué, ce programme est le moyen le plus simple pour réaliser la routine fill. Cette routine est cependant trop longue et trop lente.

Le moyen le plus rapide consiste à utiliser les instructions de chargement de bloc. Pour remplir une zone, il suffit d'utiliser ces instructions volontairement de façon incorrecte (voir chapitre 4.3.).

Programme source :

(Lignes 10-70 comme ci-dessus)

```
80 'sbc hl,de ;longueur de bloc
90 'ld b,h ;charger longueur bloc
100 'ld c,l ;dans compteur d'octets
110 'ld h,d ;charger adresse debut
120 'ld l,e ;dans debut bloc source (HL)
130 'inc de ;adresse objet=adresse debut+1
140 'ld (hl),a ;charger valeur dans premier octet source
150 'ldir
160 'ret
170 'end
```

Traduisez également ce programme pour un chargeur Basic. Lancez le programme, choisissez l'adresse de début 49152, l'adresse de fin 53247 et la valeur &FF (entrez ces valeurs en décimal).

Le bloc se trouve dans la zone de l'écran. La valeur=255=&x1111 1111 correspond à 8 points mis. Vous devriez obtenir comme résultat sur l'écran des raies larges de 2 points.

Routines de transfert

Nous allons maintenant utiliser les instructions de chargement de bloc "correctement", pour écrire une routine de transfert. Ce programme doit transférer une zone de la mémoire dans un autre emplacement. A l'aide d'un programme Basic, les adresses de début et de fin du bloc source seront entrées ainsi que l'adresse de début du bloc objet. Ce programme testera également si les valeurs entrées sont correctes. Pour la transmission des paramètres, nous utiliserons les adresses suivantes :

Bloc source	Début :	&A020/&A021
Bloc source	Fin :	&A022/&A025
Bloc objet	Début :	&A024/&A025

L'adresse de début du programme en langage-machine est alors &A026. Si les blocs source et objet ne se chevauchent pas, le bloc objet

doit contenir les données correctes, même si cela a pour effet d'effacer l'ancien contenu du bloc source.

Programme source :

```
5 'ORG &A026
10 'QANF EQU &A020 ;ADRESSE DEBUT BLOC SOURCE
20 'QEND EQU &A022 ;ADRESSE FIN BLOC SOURCE
30 'ZANF EQU &A024 ;ADRESSE DEBUT BLOC OBJET
40 ' ;DEBUT PROGRAMME, DETERMINER LONGUEUR BLOC
50 'LD HL,(QEND)
60 'LD DE,(QANF)
70 'OR A ;SUPPRIMER CARRY POUR SBC
80 'SBC HL,DE ;LONGUEUR DE BLOC -1
90 'INC HL ;+1=LONGUEUR DE BLOC
100 'LD B,H ;STOCKER LONGUEUR DE BLOC
110 'LD C,L ;DANS BC
115 ' ;DECIDER SI INC- OU DECREMENTATION
120 'LD HL,(QANF)
130 'SBC HL,DE ;ZANF INFERIEUR A
140 'JR C,LADINC ;QANF, ALORS LADINC
150 'SBC HL,BC ;DIFFERENCE SUPERIEURE A
160 'JR NC,LADINC ;LONGUEUR DE BLOC, ALORS LADINC
170 'LD HL,(ZANF)
180 'ADD HL,BC ;ZANF+LONGUEUR
190 'DEC HL ;-1=FIN DE BLOC OBJET
200 'EX DE,HL ;CHARGER DE HL DANS DE
210 'LD HL,(QEND) ;FIN BLOC SOURCE
220 'LDDR
230 'RET
240 ' ;INCREMENTER CHARGEMENT DE BLOC
250 'LADINC EX DE,HL ;DEBUT SOURCE DE DE DANS HL
260 'LD DE,(ZANF)
270 'LDIR
280 'RET
290 'END
```

Le début et la fin du programme ne nécessitent pas d'explication supplémentaire. La partie centrale où on décide s'il s'agit d'utiliser l'instruction LDDR ou l'instruction LDIR pose plus de

problèmes (lignes 115-160). Essayez de bien comprendre la nécessité de cette décision (chapitre 2.3). Normalement, donc si les blocs ne se chevauchent pas, nous utiliserons l'instruction LDIR. Si l'adresse de début objet est plus petite que l'adresse de début du bloc source, LDIR peut également être utilisé. La soustraction en ligne 130 et le saut en ligne 140 permettent le saut à "incrémentation de chargement de bloc" pour le cas ou Zanf<Qanf. Si Zanf>=Qanf, il faut décider si Zanf<=Qend.

```
              Zanf <= Qend
              Zanf <= Qanf+Longueur-1
Zanf-Qanf-Longueur <= -1
              HL-BC <= -1
```

Si le Carry est mis après HL-BC (résultat inférieur ou égal à -1), c'est LDDR qui doit être utilisé. Si le Carry=0, HL-BC était donc >=0, donc Zanf ne se trouvait pas dans le bloc source et on saute à LDIR.

Pour intégrer ce programme dans le moniteur, il faut le placer en lignes DATA. Pour un programme de cette dimension le risque est si grand que des erreurs se glissent dans les lignes DATA. Il y a deux possibilités de remédier à ce problème. Pendant la lecture des lignes DATA, on peut additionner les valeurs lues en DATA et la somme ainsi obtenue est comparée à une checksum. Si la somme finale ne correspond pas à la checksum, c'est qu'il y a une erreur. Pour notre programme, cela se présente ainsi :

```
10 FOR I=&A020 TO &A051
20 READ a$:a=VAL("&"+a$):POKE 1,a:s=s+a:NEXT
30 DATA 00,80,FF,bF,00,c0
40 DATA 2A,22,A0,ED,5B,20,A0,B7
50 DATA ED,52,23,44,4D,2A,24,A0
60 DATA ED,52,38,10,ED,42,30,0C
70 DATA 2A,24,A0,09,2B,EB,2A,22
80 DATA A0,ED,B8,C9,EB,ED,5B,24
90 DATA A0,ED,B0,C9
```

Et la ligne 100 comparera la somme finale à la checksum :

```
100 IF s<>5186 THEN PRINT"Erreur en DATAs" ELSE PRINT "ok!"
```

La seconde possibilité est cependant plus simple à mettre en oeuvre pour nous :

Après le listing assembleur, vous avez la possibilité de stocker le code objet sur cassette ou sur disquette. Ce programme peut être ensuite rechargé avec l'instruction >LOAD "nom du programme"<, y compris à partir d'un programme Basic.

Un programme de moniteur doit aussi offrir la possibilité de charger et de sauvegarder des programmes en langage-machine.

Ceci peut être aisément réalisé avec

```
LOAD "nom",adresse
```

et

```
SAVE "nom",B,adresse de début,longueur
```

Si vous reliez toutes ces fonctions entre elles, votre moniteur "connaîtra" alors les instructions suivantes :

M-(Memory)	- montrer le contenu de la mémoire
F-(Fill)	- remplir une zone de mémoire d'une valeur (V)
T-(Transfer)	- transférer des zones de mémoire
L-(Load)	- charger un programme en langage-machine
S-(Save)	- sauvegarder un programme en langage-machine

Routine compare

Intéressons-nous maintenant à la routine de comparaison. Elle permet de comparer entre elles deux zones de la mémoire. Son nom d'instruction abrégé est C. Il faut entrer à partir de programmes Basic les adresses de début et de fin du premier bloc et l'adresse de début du bloc avec lequel doit se faire la comparaison. Toutes les adresses du second bloc dont le contenu est différent de celui des adresses correspondantes du premier bloc sont affichées.

```
10 'org &a060
20 'flag DB 1
30 'Deb DS 2
40 'Fin DS 2
50 'Debcom DS 2 ;debut bloc comparaison
60 'ld de,(Deb)
70 'ld hl,(Fin)
80 'or a
90 'sbc hl,de ;longueur de bloc
100 'inc hl ;+1
110 'ld b,h
120 'ld c,l ;charger dans bc
130 'ex de,hl ;deb dans hl
140 'ld de,(debcom) ;pointeur de bloc
150 'encore ld a,(de) ;element de comparaison
160 'inc de
170 'cpi ;comparer (hl) a A
180 'jr nz,sortie ;different alors sortie
190 'jp pe,encore ;element suivant
200 'ld a,b
210 'ld (flag),a ;fin,flag=0
220 'ret
230 'sortie ld (deb),hl
240 'ld (debcom),de
250 'ret pe ;pas encore fin de bloc
260 'dec b ;b=255
270 'ld a,b
280 'ld (flag),a ;flag=255
290 'ret ;fin de bloc
300 'end
```

Les lignes 20-50 réservent de la place en mémoire pour les données à transmettre. On utilise les pseudo-instructions à cet effet. L'instruction DB (Define Byte) place la valeur fournie comme opérande à l'adresse actuelle. En l'occurrence, la valeur 1 est donc placée à l'adresse &A060. Cette case mémoire fait office de flag pour la communication avec le programme Basic. Pour les flags, les valeurs suivantes sont possibles :

1 - Une différence a été constatée au cours de la comparaison, masi celle-ci n'est pas encore achevée.

0 - La comparaison est achevée

255 - La comparaison est achevée et une différence a été constatée pour le dernier élément du bloc.

En lignes 30 et 50, nous avons utilisé la pseudo-instruction DS (Define Strorage : définir place mémoire). La pseudo-instruction DS renvoie l'assembleur vers le mpc, pour augmenter ainsi le nombre de cases mémoire indiqué. Cet emplacement de la mémoire est ainsi tenu libre et nous pourrons y stocker des variables de transmission de paramètres. En l'occurrence, nous avons besoin pour Deb, Fin et Debcom de 2 octets chaque fois. Nous avons donc utilisé l'instruction DS 2.

En lignes 60-120, la longueur du premier bloc est chargée dans le compteur d'octets (Byte Counter) BC. L'instruction INC HL en ligne 100 est nécessaire car sinon le dernier élément échapperait à la comparaison. En ligne 130, l'adresse de début du premier bloc est chargée dans HL et en ligne 140 l'adresse de début du bloc à comparer est chargée dans DE.

En 150 commence la boucle principale du programme. Les valeurs successives du bloc de comparaison sont d'abord chargées dans l'accumulateur (150) et le pointeur dans le bloc de comparaison est augmenté (160). CPI a plusieurs fonctions. Il compare le contenu de l'accumulateur (=valeur dans le bloc à comparer) avec la valeur à l'adresse HL (=valeur dans le premier bloc). Le flag Z est modifié en fonction du résultat de la comparaison. HL est ensuite augmenté et BC est diminué. Si BC est ensuite à zéro, le P/V est annulé (PO), sinon il est mis.

En ligne 180, on saute à la sortie si les valeurs comparées se sont révélées différentes. S'il y avait identité, la ligne 190 répète la boucle si P/V=0, c'est-à-dire si PE. Si P/V égale par contre 1, le flag est mis à 0 en ligne 200 et on retourne au Basic.

En ligne 220 commence la section de programme de sortie.

Les pointeurs de bloc actuels sont d'abord sauvegardés. DE contient alors l'adresse de la case mémoire qui est différente, augmentée de 1. Après que cette adresse ait été sortie par le programme Basic, la routine est à nouveau appelée et poursuivie à l'adresse qui convient, puisque Deb et Debcom ont été fixés sur l'état actuel avant le retour au Basic. Si la comparaison du bloc n'est pas encore achevée, c'est-à-dire si BC<>0 et P/V=1, donc si PE, l'instruction RET est exécutée. Si par contre le dernier élément a été comparé (pas d'identité), les lignes 260/280 fixent Flag (V) sur 255 pour qu'il soit possible de faire la distinction avec la cas où la fin du bloc a été atteinte et où il y avait identité (c'est-à-dire Flag=0).

```
A060            10         ORG  &a060
A060  01        20 FLAG    DB   1
A061            30 DEB     DS   2
A063            40 FIN     DS   2
A065            50 DEBCOM  DS   2 ;debut bloc comparaison
A067  ED5B61A0  60         LD   de,(Deb)
A06B  2A63A0    70         LD   hl,(Fin)
A06E  B7        80         OR   a
A06F  ED52      90         SBC  hl,de ;longueur de bloc
A071  23       100         INC  hl ;+1
A072  44       110         LD   b,h
A073  4D       120         LD   c,l ;charger dans bc
A074  EB       130         EX   de,hl ;deb dans hl
A075  ED5B65A0 140         LD   de,(debcom) ;pointeur de bloc
A076  1A       150 ENCORE  LD   a,(de) ;element de comparaison
A07A  13       160         INC  de
A07B  EDA1     170         CPI   ;comparer (hl) a A
A07D  20FE     180         JR   nz,sortie ;different alors
                                sortie
A07F  EA79A0   190         JP   pe,encore ;element suivant
A082  78       200         LD   a,b
A083  326A0    210         LD   (flag),a ;fin, flag=0
A086  C9       220         RET
**** Ligne 180 : SORTIE=&A087
A087  2261A0   230 SORTIE  LD   (deb),hl
```

```
A08A  ED5365A0 240          LD   (debcom),de
A08E  E8       250          RET  pe ;pas encore fin de bloc
A08F  05       260          DEC  b ;b=255
A090  78       270          LD   a,b
A091  3260A0   280          LD   (flag),a ;flag=255
A094  C9       290          RET   ;fin de bloc
```

```
Programme : Compare
Debut : &A060        Fin : &A094
Longueur : 0035
  O  Erreur
Table de variables :
FLAG  A060 DEB  A061  FIN  A063  DEBCOM  A065  ENCORE  A079
SORTIE A087
```

Le programme Basic pour appeler la routine se présente ainsi :

```
10 REM compare
20 MEMORY &9FFF
30 MODE 2
40 POKE &A060,1
50 INPUT"debut de bloc :&",a$
60 adr=&A061:GOSUB 170
70 INPUT"Fin de bloc :&",a$
80 adr=&A063:GOSUB 170
90 INPUT"debut du bloc a comparer :&",a$
100 adr=&A065:GOSUB 170
110 CALL &A067
120 w=PEEK(&A060)
130 IF w=0 THEN END
140 PRINT HEX$(PEEK(&A061)+256*PEEK(&A062)-1,4)
150 IF w=1 THEN 110
160 END
170 a=VAL("&"+a$)
180 IF a<0 THEN a=A+2^16
190 ah=INT(a/256)
200 POKE adr,a-ah*256
210 POKE adr+1,ah
220 RETURN
```

Vous pouvez aisément programmer vous-même l'instruction GO (G) qui permet d'appeler un programme en langage-machine à partir du programme de moniteur (par exemple pour effectuer des tests). Il vous suffit d'utiliser l'instruction Basic >CALL adresse< avec une routine d'entrée pour l'adresse (V).

Dans le programme Compare, les allers et retours entre Basic et langage-machine sont peu pratiques. Il était cependant nécessaire d'utiliser concurremment le Basic et le langage-machine puisque nous ne savons pas encore programmer les entrées-sorties (INPUT et PRINT) en langage-machine. Il est assez difficile de réaliser des routines d'entrée/sortie en Basic. Pour sortir par exemple une lettre sur l'écran, il faudrait fournir la position exacte de la lettre, en tenant compte de la différence dûe au scrolling. Il faut ensuite lire dans la mémoire des caractères les 8 octets qui servent à représenter un caractère (ROM &3800 à &3FFF) et les écrire dans la mémoire écran. Mais comme la sortie de caractères sur l'écran fonctionne dès la mise sous tension de l'ordinateur, c'est que la routine nécessaire à cet effet doit déjà se trouver en ROM. Si nous connaissions cette routine ou au moins l'adresse à laquelle elle commence, nous pourrions l'appeler directement à partir de notre programme en langage-machine. L'utilisation de ce qu'on appelle les routines système constitue une technique très utile et très intéressante.

6. Utilisation de routines système

6.1. LE DESASSEMBLEUR

Le CPC possède une ROM 32 K. Ces 32 K contiennent des routines système. La ROM des 16 K supérieurs (&C000 à &FFFF) contient le Basic, la ROM des 16 K inférieurs (&0 à &3FFF) le système d'exploitation de l'ordinateur. Le système d'exploitation contient un grand nombre de routines qui peuvent intéresser les programmeurs en langage-machine.

Pour pouvoir analyser ces routines, il nous faut un outil supplémentaire, le désassembleur.

Un désassembleur interprète les octets d'une zone indiquée comme code machine et convertit les nombres en instructions assembleur correspondantes. Le désassembleur est ainsi le complément de l'assembleur. Le désassembleur nous permet de reconvertir en instructions assembleur, après les avoir chargés, des programmes en langage-machine dont nous ne possédons pas le source, par exemple des programmes en langage-machine fournis sous forme de lignes de DATA. Nous pouvons traduire ainsi également les routines internes de l'ordinateur. Ces programmes qui ont été réalisés par des professionnels peuvent nous apprendre beaucoup de choses. En outre, nous pouvons également utiliser ces routines dans nos propres programmes.

Pour tester le désassembleur, lancez-le avec >RUN<. Pour un CPC 464, entrez &BACB comme adresse de début et &BADB comme adresse de fin. Pour un CPC 6128 ou 664, entrez &BAC6 pour le début et &BAD6 pour la fin.

Vous devez obtenir une image à peu près semblable à celle-là (cela dépend du modèle de CPC en votre possession) :

```
BACB    F3          DI
BACC    D9          EXX
BACD    59          LD      E,C
BACE    CB D3       SET     2,E
BAD0    CB DB       SET     3,E
BAD2    ED 59       OUT     (C),E
BAD4    D9          EXX

BAD5    7E          LD      A,(HL)
BAD6    D9          EXX
BAD7    ED 49       OUT     (C),C
BAD9    D9          EXX
BADA    FB          EI
BADB    C9          RET
```

En entrant le programme de désassembleur, faites attention à la description du programme !

```
10 MEMORY &9FFF
20 MODE 2
30 GOTO 990
40 LOCATE 18,4:PRINT"D E S A S S E M B L E R   Z 8 0"
50 LOCATE 5,7:INPUT"Imprimante (o/n) ",e$
60 IF e$="o" THEN aus=8 ELSE aus=0
70 LOCATE 5,10:INPUT"Adresse de debut : &",a$
80 GOSUB 900:anfa=a
90 LOCATE 5,12:INPUT"Adresse de fin   : &",a$
100 GOSUB 900:ende=a
110 IF anfa>ende THEN 20
120 pc=anfa
```

```
130 MODE 2
140 adr=pc
150 PRINT#aus,HEX$(adr,4);"  ";
160 iflag=0
170 GOSUB 940
180 GOSUB 300
190 IF iflag THEN 600
200 IF w=&CF OR w=&D7 OR w=&DF OR w=&EF THEN pr$=pr$+" /DW:nn"
210 IF INSTR(pr$,"n")<>0 THEN 700
220 IF INSTR(pr$,"e")<>0 THEN 820
230 po=INSTR(pr$," ")
240 IF PR$="" THEN PR$="???"
250 IF po=0 THEN PRINT#aus,TAB(21);pr$;:GOTO 270
260 PRINT#aus,TAB(21);LEFT$(pr$,po-1);TAB(27);RIGHT$(pr$,LEN(pr$)-po);
270 PRINT#aus
280 IF pc<=ende THEN 140
290 END
300 REM Interpreter
310 IF (w=&DD OR w=&FD) AND NOT iflag THEN 490
320 IF w=&ED THEN 460
330 IF w=&CB THEN 410
340 GOSUB 540
350 ON col GOTO 370,390,360
360 pr$=bef$(w):RETURN
370 IF w=&76 THEN pr$="HALT":RETURN
380 pr$="LD "+regtab$(co2)+","+reg$:RETURN
390 IF co2=0 OR co2=1 OR co2=3 THEN a$=" A," ELSE a$=" "
400 pr$=arilog$(co2)+a$+reg$:RETURN
410 REM cb
420 GOSUB 940
425 IF iflag THEN dis=w:GOSUB 940
430 GOSUB 540
440 IF col=0 THEN pr$=rotschi$(co2)+" "+reg$ ELSE pr$=bitti$(col)+STR$
(co2)+","+reg$
450 RETURN
460 REM ed
470 GOSUB 940
480 IF w<&40 OR w>&BF THEN pr$="???":RETURN ELSE GOTO 360
```

```
490 REM xy
500 iflag=-1
510 IF w=&DD THEN i$="IX" ELSE i$="IY"
520 GOSUB 940
530 GOTO 300
540 REM decomposer code
550 co1=(w AND &X11000000)/64
560 co2=(w AND &X111000)/8
570 co3=w AND &X111
580 reg$=regtab$(co3)
590 RETURN
600 REM indexe
610 po=INSTR(pr$,"HL")
620 IF po=0 THEN pr$="???":GOTO 230
630 IF INSTR(pr$,"(HL)")<>0 THEN 670
640 IF pr$="EX DE,HL" THEN pr$="???":GOTO 230
650 IF pr$="ADD HL,HL" THEN pr$="ADD "+i$+","+i$:GOTO 230
660 pr$=LEFT$(pr$,po-1)+i$+RIGHT$(pr$,LEN(pr$)-po-1):GOTO 200
670 IF LEFT$(pr$,2)="JP" THEN 660
680 IF pc-adr<3 THEN GOSUB 940:dis=w
685 IF dis>127 THEN dis$=STR$(dis-256) ELSE dis$="+"+RIGHT$(STR$(dis),
LEN(STR$(dis))-1)
690 i$=i$+dis$:GOTO 660
700 REM remplacer n
710 po=INSTR(pr$,"nn")
720 IF po<>0 THEN 770
730 po=INSTR(pr$,"n")
740 GOSUB 940
750 pr$=LEFT$(pr$,po-1)+"&"+HEX$(w,2)+RIGHT$(pr$,LEN(pr$)-po)
760 GOTO 230
770 GOSUB 940:lb=w
780 GOSUB 940
790 wert=w*256+lb
800 pr$=LEFT$(pr$,po-1)+"&"+HEX$(wert,4)+RIGHT$(pr$,LEN(pr$)-po-1)
810 GOTO 230
820 REM remplacer e
830 po=INSTR(pr$,"e")
840 GOSUB 940
```

```
850 IF w>127 THEN w=w-256:REM 2er-Komp.
860 w=w+2
870 a$="$"+STR$(w)+" >"+"&"+HEX$(pc+w-2,4)
880 pr$=LEFT$(pr$,po-1)+a$+RIGHT$(pr$,LEN(pr$)-po)
890 GOTO 230
900 REM Conversion hex -> dec
910 IF a$="" THEN a=0:RETURN
920 a=VAL("&"+a$):IF a<0 THEN a=a+65536
930 RETURN
940 REM Lire octet
950 w=PEEK(pc)
960 pc=pc+1
970 PRINT#aus,HEX$(w,2);" ";
980 RETURN
990 REM init
1000 DIM regtab$(7),rotschi$(8),bitti$(3),arilog$(7),bef$(255)
1010 FOR i=0 TO 7:READ regtab$(i):NEXT
1020 FOR i=0 TO 7:READ rotschi$(i):NEXT
1030 FOR i=1 TO 3:READ bitti$(i):NEXT
1040 FOR i=0 TO 7:READ arilog$(i):NEXT
1050 FOR i=0 TO &7F:READ bef$(i):NEXT
1060 FOR i=&80 TO &9F:bef$(i)="":NEXT
1070 FOR i=&A0 TO &FF:READ bef$(i):NEXT
1080 GOTO 40
1090 REM DATAS
1100 DATA B,C,D,E,H,L,(HL).A
1110 DATA RLC,RRC,RL,RR,SLA,SRA,???,SRL
1120 DATA BIT,RES,SET
1130 DATA ADD,ADC,SUB,SBC,AND,XOR,OR,CP
1140 DATA NOP,"LD BC,nn","LD (BC),A",INC BC,INC B,DEC B,"LD B,n",RLCA
1150 DATA "EX AF,AF'","ADD HL,BC","LD A,(BC)",DEC BC,INC C,DEC C,"LD C
,n",RRCA
1160 DATA DJNZ e,"LD DE,nn","LD (DE),A",INC DE,INC D,DEC D,"LD D,n",RL
A
1170 DATA JR e,"ADD HL,DE","LD A,(DE)".DEC DE,INC E,DEC E,"LD E,n",RRA
1180 DATA "JR NZ,e","LD HL,nn","LD (nn),HL",INC HL,INC H,DEC H,"LD H,n
",DAA
1190 DATA "JR Z.e","ADD HL,HL","LD HL,(nn)",DEC HL,INC L,DEC L,"LD L,n
```

```
",CPL
1200 DATA "JR NC,e","LD SP,nn","LD (nn),A",INC SP,INC (HL),DEC (HL),"L
D (HL),n",SCF
1210 DATA "JR C,e","ADD HL,SP","LD A,(nn)",DEC SP,INC A,DEC A,"LD A,n"
,CCF
1220 DATA "IN B,(C)","OUT (C),B","SBC HL,BC","LD (nn),BC",NEG,RETN,IM
0,"LD I,A"
1230 DATA "IN C,(C)","OUT (C),C","ADC HL,BC","LD BC,(nn)",,RETI,,"LD R
,A"
1240 DATA "IN D,(C)","OUT (C),D","SBC HL,DE","LD (nn),DE",,,IM 1,"LD A
,I"
1250 DATA "IN E,(C)","OUT (C),E","ADC HL,DE","LD DE,(nn)",,,IM 2,"LD A
,R"
1260 DATA "IN H,(C)","OUT (C),H","SBC HL,HL","LD (nn),HL",,,,RRD
1270 DATA "IN L,(C)","OUT (C),L","ADC HL,HL","LD HL,(nn)",,,,RLD
1280 DATA ,,"SBC HL,SP","LD (nn),SP",,,,
1290 DATA "IN A,(C)","OUT (C),A","ADC HL,SP","LD SP,(nn)",,,,
1300 DATA LDI,CPI,INI,OUTI,,,,,LDD,CPD,IND,OUTD,,,,
1310 DATA LDIR,CPIR,INIR,OTIR,,,,,LDDR,CPDR,INDR,OTDR,,,,
1320 DATA RET NZ,POP BC,"JP NZ,nn",JP nn,"CALL NZ,nn",PUSH BC,"ADD A,n
",RST &00
1330 DATA RET Z,RET,"JP Z,nn",->,"CALL Z,nn",CALL nn,"ADC A,n",RST &08
1340 DATA RET NC,POP DE,"JP NC,nn","OUT (n),A","CALL NC,nn",PUSH DE,"S
UB n",RST &10
1350 DATA RET C,EXX,"JP C,nn","IN A,(n)","CALL C,nn",->,"SBC A,n",RST
&18
1360 DATA RET PO,POP HL,"JP PO,nn","EX (SP),HL","CALL PO,nn",PUSH HL,"
AND n",RST &20
1370 DATA RET PE,JP (HL),"JP PE,nn","EX DE,HL","CALL PE,nn",->,"XOR n"
,RST &28
1380 DATA RET P,POP AF,"JP P,nn",DI,"CALL P,nn",PUSH AF,"OR n",RST &30
1390 DATA RET M,"LD SP,HL","JP M,nn",EI,"CALL M,nn",->,"CP n",RST &38
```

EXPLICATION :

Lignes 10-130 :

Menu : Entrée des adresses de début et de fin. Choix entre imprimante et écran.

Lignes 140-290 :

Boucle principale du programme :

Ligne 150 :

L'adresse actuelle est ici sortie.

Ligne 170 :

Lire et sortir le prochain octet.

Ligne 180 :

Saut au sous-programme qui effectue l'interprétation.

Ligne 190 :

Sauter au traitement des instructions indexées si iflag est mis (=-1).

Ligne 200 :

Traitement des instructions RST qui utilisent le prochain mot de données.

Ligne 210 :

Branchement si l'instruction contient des nombres.

Ligne 220 :

Branchement si l'instruction contient des distances relatives.

Lignes 230-270 :

Sortie formatée.

Ligne 280 :

Si pas encore fini, alors retour au début de la boucle principale.

Lignes 300-350 :

Sous-programme d'interprétation.

Ligne 310 :

Saut au traitement des instructions indexées.

Ligne 320 :

Saut au traitement des instructions qui commencent par &ED.

Ligne 330 :

Saut au traitement des instructions qui commencent par &CB.

Ligne 340 :

Saut au sous-programme qui décompose w en co1 (bits7,6), co2 (bits 5-3) et co3 (bits 2-0)

Ligne 350 :

Si co1=0 et co1=3, aller en ligne 360, c'est-à-dire lire instructions dans la table, sinon en ligne 370, instructions de la forme LD reg,reg ou en ligne 390, instructions arithmétiques et logiques sur 8 bits.

Ligne 360 :

Déterminer pr$ à partir de la table.

Lignes 370/380 :

Instructions de la forme LD, r,r' et HALT

Lignes 390/400 :

Instructions ari-log

Lignes 410-450 :

Traitement des instructions qui commencent par le code &CB

Ligne 420 :

Lire le prochain octet

Ligne 430 :

Décomposer en co1, co2, co3

Ligne 440 :

Produire pr$ pour instructions de rotation ou de décalage (co1=0) et instructions de manipulation de bits.

Lignes 460-480 :

Traitement des instructions qui commencent par le code &ED

Ligne 470 :

Lire l'octet suivant .

Ligne 480 :

Si code valable, déterminer dans table (ligne 360)

Lignes 490-530 :

Premier traitement des instructions indexées (de la ligne 310)

Ligne 500 :

Mettre flag

Ligne 510 :

Sauvegarder le registre dans i$ (soit IX soit IY)

Ligne 520 :

Lire le prochain octet

Ligne 530 :

Recommencer l'interprétation (ligne 300)

Lignes 540-590 :

Sous-programme pour décomposer le code ; w est décomposé en co1 (bits 7,6), co2 (bits 5-3) et co3 (bits 2-0). reg$ contient le registre appartenant à co3

Lignes 600-690 :

Deuxième traitement des instructions indexées (saut de la ligne 190). Vérifier si instruction indexée autorisée ; si oui, HL est remplacé par i$. Si indication de distance nécessaire, les lignes 680/690 lisent la distance.

Lignes 700-810 :

Si pr$ contient un "n", n est ici remplacé par un nombre

Lignes 730-760 :

Nombres sur 1 octet (n)

Lignes 770-810 :

Nombres sur 2 octets (nn)

Lignes 820-890 :

Remplacer offset (e)

Lignes 850/860 :

Calculer offset

Lignes 870/880 :

Remplacer offset

Lignes 900-930 :

Sous-programme de conversion hexa - dec

Lignes 940-980 :

Sous-programme lire et sortir prochain octet

Lignes 990-1080 :

Initialisation : création des tables

Lignes 1090-1380 :

Lignes DATA

Liste des variables :

a	Renvoyé par le SUB "hex-dec". Valeur de a$ interprétée comme nombre hexa
a$	Entrée d'un nombre hexa/ transmis au SUB "hexa-dec"
adr	Adresse du premier code de l'instruction actuelle

anfa	Adresse de début de l'interprétation
aus	Canal du périphérique de sortie
col	Bits 7 et 9
co2	Bits 5 à 3
co3	Bits 2 à 0
dis	Distance pour les instructions indexées
e$	Chaîne d'entrée (o/n)
ende	Interpréter adresse de fin
i$	Contient le registre d'index actuel
iflag	Mis si adressage indexé, sinon annulé
lb	Stockage provisoire de l'octet faible pour les nombres sur deux octets
pc	Pointeur de programme
po	Position de n, nn, e, HL ... dans pr$
pr$	Print$ contient l'instruction assembleur
reg$	Registre : renvoyé par le SUB décomposer code
w	reg$ contient le registre attribué à co3
wert	Valeur d'un nombre sur deux octets (nn)

Tableaux

regtab$	Registres
rotschie$(7)	Instructions de rotation et de décalage
bitti$(3)	Instructions de manipulation de bits
arilog$(7)	Instructions arithmétiques ou logiques
bef$(255)	0 à &3F : instructions qui commencent par &ED et qui ont le numéro comme premier octet &40-&BF : instructions qui commerncent par &ED et qui ont le numéro comme second octet &BF-&FF : instructions qui ont le numéro comme premier octet

6.2. ROUTINES SYSTEME

La routine de sortie d'un caractère sur l'écran est certainement l'une des routines système les plus importantes. Elle peut être appelée avec CALL &BB5A. Cette routine sort le caractère qui correspond à la valeur se trouvant dans l'accumulateur.

Ecrivez un programme pour sortir le jeu de caractères (code 32 à 255) du CPC.

Solution :

```
10 'org &a000
20 'print equ &bb5A
30 'ld b,223 ;compteur=255-32
40 'ld a, 32
50 'boucle call print ;sortir chr$(a)
60 'inc a
70 'djnz boucle
80 'ret
90 'end
```

Pour la sortie de caractères, l'assembleur offre la pseudo-instruction DM. L'instruction DM est suivie d'un mot entre guillemets. Les codes ASCII des lettres du mot sont placés par DM à partir de l'adresse actuelle. Voyez par exemple le programme suivant :

```
10  'org &a000
20  'print equ &bb5A
30  'ld hl,mot ;adresse du mot a sortir
40  'boucle ld a,(hl) ;charger code ASCII de chaque lettre
    dans accu
50  'inc hl ;fixer pointeur sur lettre suivante
60  'call print
70  'or a ;fixer flags
80  'jr nz ;pas encore 0, alors lettre suivante
90  'ret
100 'mot dm "Amstrad"
110 'db 0
120 'end
```

L'octet zéro produit par DB 0 à la fin du mot (ligne 110) permet d'identifier la fin du mot à sortir.

Désassemblez la routine à partir de &BB5A. Vous obtenez :

```
pour CPC 464 :           BB5A CF 00 94   RST  &08/DW: &9400
pour CPC 664 et 6128 :   BB5A CF FE 93   RST  &08/DW: &93FE
```

En adresse &BB5A se trouve une instruction Restart à l'adresse &0008. Continuons le désassemblage :

```
pour CPC 464 :           0008 C3 82 B9   JP   &B982
pour CPC 664 et 6128 :   0008 C3 8A B9   JP   &B98A
```

Cette routine (en &B982 pour 464 et en &B98A pour 664 et 6128) est appelée "routine RST &08". Elle a pour effet que les deux octets (low/high) figurant après l'instruction RST &08 sont traités de façon spéciale. C'est pourquoi le désassembleur sort les octets suivant le RST &08 avec la marque DW (DATA-Word, c'est-à-dire octets faible et fort). DW représente également une pseudo-instruction qui a pour effet que le mot de donnée suivant l'instruction, donc un nombre sur deux octets, est placé en mémoire dans l'adresse qui convient. Les bits 0-13 sont interprétés comme adresse de ligne de saut (14 bits permettent de représenter des adresses comprises entre &0 et &3FFF. Les bits 14 et 15 servent à sélectionner entre ROM et RAM.

Le bit 14 détermine l'état de la zone &0-&3FFF. Le bit 15 détermine l'état de la zone &C000 à &FFFF (RAM écran ou ROM Basic). Un bit mis sélectionne la RAM, un bit annulé sélectionne la ROM. Quelle est la configuration et quel est le but du saut pour l'instruction suivante ?

Pour un 464 :

```
RST &08
DW  &9400
```

Pour un 664 ou 6128 :

```
RST &08
DW  &93FE
```

Décomposons :

Pour CPC464 : &9400=&8000+&1400=&X10 01 0100 0000 0000
CPC664 et 6128 : &93FE=&8000+&13FE=&X10 01 0011 1111 1110

Bit 15=1 => RAM écran
Bit 14=0 => ROM système d'exploitation
Adresse : &1400 pour un 464
Adresse : &13FE pour un 664 et 6128

Les routines AST &08/DW &9400 pour un 464 et RST &08/DW &93FE pour un 664 ou 6128 ont pour effet un saut à la routine du système d'exploitation à l'adresse &1400. La RAM écran est sélectionnée.

Bien que les instructions RST soient en principe des sauts à des sous-programmes, c'est-à-dire que l'adresse de retour est placée sur la pile, l'instruction RST &08 n'est pas un sous-programme mais un saut normal. Ceci est obtenu par manipulation de pile dans la routine à partir de &B982 pour un 464 ou &B98A pour un 664 ou 6128. Les autres instructions RST ont également des fonctions particulières. Nous les traiterons dans le cours de ce chapitre.

En nous aidant de la routine Print, nous allons maintenant écrire un programme de moniteur.

Le moniteur

Comme nous allons sortir le contenu de la mémoire sous forme de nombres hexa, nous avons d'abord besoin d'un sous-programme qui sorte un octet sous forme d'un nombre hexa. L'octet à sortir sera transmis par l'accumulateur.

Exemple : A= 63 = &3F = &X 0011 1111

F correspond aux 4 bits inférieurs (quartet inférieur)
3 correspond aux 4 bits supérieurs (quartet supérieur)

Le quartet supérieur est sorti d'abord. A cet effet, nous décalons le contenu de l'accumulateur 4 fois vers la droite (rotation sur 8

bits). Le résultat de ce décalage est &X 1111 0011. Puis nous annulons avec AND les 4 bits supérieurs. L'accumulateur contient ensuite la valeur &X 0000 0011=3. C'est cette valeur de 3 qui doit être sortie.

Le code ASCII de 3 est 51. Pour obtenir la valeur 51 dans l'accumulateur, il nous faut ajouter 48 au contenu de l'accumulateur (=3). Puis nous appelons la routine PRINT. Pour sortir le quartet faible, il nous suffit d'annuler les 4 bits supérieurs de l'ancien contenu de l'accumulateur. Après avoir ajouté 48, nous obtenons 63. Mais nous voulons obtenir F (&F=15) en sortie. La valeur ASCII de F est 70. Cela signifie que lorsque le chiffre hexa à sortir est plus grand que 9 et qu'il s'agit donc d'une lettre, il faut encore ajouter 7 avant d'appeler la routine de sortie.

Essayez d'écrire cette routine de sortie d'un octet en forme hexadécimale.

```
A000          10           ORG   &a000
A000          20  PRINT    EQU   &bb5a
A000          30                  ; sortie hexa
A000          40                  ; sort le contenu de
                                  l'accu en hexa
A000          50                  ; le registre E est modifie
A000 5F       60  SORHEX   LD    e,a ;stockage provisoire accu
A001 0F       70           RRCA  ;decaler
A002 0F       80           RRCA  ;de 4 bits
A003 0F       90           RRCA  ;vers la
A004 0F      100           RRCA  ;droite
A005 E60F    110           AND   &x1111 ;annuler bits 4-7
A007 CD0000  120           CALL  conv ;sortir quartet superieur
A00A 7B      130           LD    a,e ;ancien contenu accu
A00B E60F    140           AND   &x1111 ;annuler bits 4-7
A00D CD0000  150           CALL  conv ;sortir quartet inferieur
A010 C9      160           RET   ;fin sortie hexa
A011         170                  ;routine de conversion
A011         180
;sort le chiffre hexa correspondant au contenu de l'accumula-
teur
```

```
**** Ligne 120 : CONV=&A011
**** Ligne 150 : CONV=&A011
A011 FEA0    190  CONV    CP   &a ;valeur du chiffre<10
A013 38FE    200          JR   c,chiffr ;oui alors chiffre
A015 C607    210          ADD  a,7 ;ajouter 7 pour les lettres
**** Ligne   200 : CHIFFR=&A017
A017 C630    220  CHIFFR  ADD  a,48 ;code ascii des chiffres
                               hexa
A019 CD5ABB230            CALL print
A01C C9      240          RET  ;fin conversion
End assumed


Programme : sortie hexa
Debut : &A000    Fin : &A01C
Longueur : 001D
  0 Erreur
Table de variables :
PRINT  BB5A  SORHEX  A000  CONV  A011  CHIFFR  A017
```

Avec ce programme, nous pouvons maintenant écrire en langage-machine la sortie pour le programme Compare du chapitre précédent. Reliez les deux programmes de façon à ce que la sortie des adresses soit prise en charge par la routine ci-dessus.

```
A000            10                  ;compare
A000            20          ORG  &a000
A000            30 PRINT    EQU  &bb5a
A000            40 DEB      DS   2
A002            50 FIN      DS   2
A004            60 DEBCOM   DS   2 ;debut bloc comparaison
A006  ED5B00A0  70          LD   de,(deb)
A00A  2A02A0    80          LD   hl,(Fin)
A00D  B7        90          OR   a
A00E  ED52      100         SBC  hl,de ;longueur de bloc
A010  23        110         INC  hl ;+1
A011  44        120         LD   b,h
A012  4D        130         LD   c,l ;charger dans bc
A013  EB        140         EX   de,hl ;deb dans hl
A014  ED5B04A0  150         LD   de,(debcom) ;pointeur de bloc
```

```
A018  1A      160 ENCORE  LD   a,(de) ;element de comparaison
A019  13      170         INC  de
A01A  EDA1    180         CPI   ;comparer (hl) a A
A01C  C40000  190         CALL nz,sortie ;different alors
                               sortie
A01F  EA1810  200         JP   pe,encore ;element suivant
A022  C9      210         RET   ;fin compare
A023          220
**** Ligne 190 : SORTIE=&A023
A023  D5      230 SORTIE  PUSH de ;sauver pointeur de bloc
A024  F5      240         PUSH af ;sauver flags
A025  2B      250         DEC  hl ;diminuer hl pour sortie
A026  7C      260         LD   a,h
A027  CD0000  270         CALL sorhex ;sortir octet faible
A02A  7D      280         LD   a,l
A02B  CD0000  290         CALL sorhex ;sortir octet fort
A02E  23      300         INC  hl ;retablir hl
A02F  3E20    310         LD   a,&20 ;espace
A031  CD5ABB  320         CALL PRINT
A034  F1      330         POP  af
A035  D1      340         POP  de
A036  C9      350         RET   ;fin sortie
A037          360               ;
A037          370               ; sortie hexa
A037          380               ; sort le contenu de l'accu en
                                hexa
A037          390               ; le registre E est modifié
**** Ligne 270 : SORHEX=&A037
**** Ligne 290 : SORHEX=&A037
A037  5F      400 SORHEX  LD   e,a ;stockage provisoire accu
A038  0F      410         RRCA ;decaler
A039  0F      420         RRCA ;de 4 bits
A031  0F      430         RRCA ;vers la
A03B  OF      440         RRCA ;droite
A03C  E60F    450         AND  &x1111 ;annuler bits 4-7
A03E  CD0000  460         CALL conv ;sortir quartet superieur
A041  7B      470         LD   a,e ;ancien contenu accu
A042  E60F    480         AND  &x1111 ;annuler bits 4-7
A044  CD0000  490         CALL conv ;sortir quartet inferieur
A047  C9      500         RET  ;fin sortie hexa
```

```
A048            510                     ;routine de conversion
A048            520
;sort le chiffre hexa correspondant au contenu de l'accumulateur
**** Ligne 460 : CONV=&A048
**** Ligne 490 : CONV=&A048
A048  FEOA      530 CONV       CP    &a ;valeur du chiffre<10
A04A  38FE      540            JR    c,chiffr ;oui alors chiffre
A04C  38FE      550            JR    c,chiffr
A04E  C607      560            ADD   a,7 ;ajouter 7 pour les lettres
**** Ligne 540 : CHIFFR=&A050
**** Ligne 550 : CHIFFR=&A050
A050  C630      570 CHIFFR     ADD   a,48 ;=code ascii des chiffres
                                     hexa
A052  CD5ABB    580            CALL print
A055  C9        590            RET   ;fin conversion
End Assumed

Programme : compare
Debut : &A000        Fin : &A055
Longueur : 0056
  O Erreur
Table des variables :
PRINT  BB5A DEB    A000 FIN    A002 DEBCOMA004 ENCORE A018
SORTIE A023 SORHEX A037 CONV A048 CHIFFR  A050
```

Continuons maintenant notre routine de moniteur. Les adresses de début et de fin seront entrées et transmises à partir du Basic.

```
10  'org &a000
20  'print equ &bb5a
30  'Deb DS 2
40  'Fin DS 2
```

Chargeons d'abord l'adresse de début dans HL et sortons-la :

```
50  'ld hl,(deb)
60  'sui16 ld a,h, ;sortir octet fort
70  'call sorhex
80  'ld a,l ;sortir octet faible
90  'call sorhex
```

Il faut ensuite sortir un caractère espace :

```
100 'ld a,&20 ;ASCII de l'expace
110 'call print
```

Les valeurs des 16 (8 en mode 1) cases mémoire suivantes sont ensuite sorties :

```
120 'ld b,16 ;compteur
130 'sui ld a,(hl) ;charger octet dans accu
140 'call sorhex ;et sortir
150 'ld a,&20 ;espace
160 'call print ;sortir
170 'inc hl
180 'djnz sui
```

Un espace est ensuite sorti et les 16 (8) derniers octets sont sortis sous forme de caractères ASCII. Des codes qui sont plus grand que 127, on ote 128 (le bit 7 est annulé). Pour les codes qui sont plus petits que 32 (caractères de commande), un point (ASCII=46) est sorti.

```
190 'ld a,&20
200 'call print ;espace
210 'ld de,16
220 'or a ;carry=0
230 'sbc hl,de ;diminuer pointeur de 16
240 'ld b,16
250 'suias ld a,(hl) ;charger octet dans accu
260 'inc hl ;augmenter pointeur
270 'res 7,a ;convertir caractere graphique en ASCII
280 'cp &20 ;superieur egal 32??
290 'jr nc,pr ;oui, alors sortie
300 'ld a,46 ;ASCII pour point
310 'pr call print ;sortir caractere
320 'djnz suias
```

Pour arriver au début de la prochaine ligne, les codes 13 et 10 sont envoyés :

(CHR$(13) = Carriage Return = retour de chariot)
(CHR$(10) = Line Feed = passage à la ligne suivante)

```
330 'ld a,13 ;carriage return
340 'call print ;sortir
350 'ld a,10 ;line feed
360 'call print ;sortir
```

On teste maintenant si on est déjà arrivé à la fin :

```
370 'push hl ;sauver pointeur
380 'ld de,(Fin)
390 'or a ;carry=0
400 'sbc hl, de ;pointeur-Fin<=0
410 'pop hl ;amener pointeur (pas de modification des flags!!)
420 'jr c,sui16 ;hl-de<0, alors suite
430 'jr z,sui16 ;hl-de=0, alors suite
440 'ret ;hl-de>0, alors fin
450 ';fin moniteur
460 '; sortie hexa
```

Il ne reste plus maintenant qu'à ajouter la routine Sorhex et notre programme pourra tourner :

```
470 '; sort le contenu de l'accu en hexa
480 '; le registre E est modifie
490 'sorhex ld e,a ;stockage provisoire accu
500 'rrca ;decaler
510 'rrca ;de 4 bits
520 'rrca ;vers la
530 'rrca ;droite
540 'and &x1111 ;annuler bits 4-7
550 'call conv ;sortir quartet superieur
560 'ld a,e ;ancien contenu accu
570 'and &x1111 ;annuler bits 4-7
580 'call conv ;sortir quartet inferieur
590 'ret ;fin sortie hexa
600 ';routine de conversion
610 ';sort le chiffre hexa correspondant au contenu de
l'accumulateur
```

```
620 'conv cp &a ;valeur du chiffre<10
630 'jr c,chiffr ;oui alors chiffre
640 'add a,7 ;ajouter 7 pour les lettres
650 'chiffr add a,48 ;=code ascii des chiffres hexa
660 'call print
670 'ret ;fin conversion
680 'end
```

Cette routine ne nous permet cependant de lire que la RAM de l'ordinateur. Pour accéder à la ROM, nous utiliserons l'instruction RST &18. Cette instruction effectue sur le CPC ce qu'on appelle un Far Call. Les deux octets suivant le RST &18 constituent un pointeur sur l'adresse d'un vecteur de saut. A l'adresse de vecteur indiquée se trouvent 3 octets. Les deux premiers octets indiquent l'adresse de saut et le troisième octet détermine la configuration ROM/RAM.

Exemple :

```
&A000    RST &18
&A001    DW  vecadr
&A003    RET

Vecadr   DW adrobj
         DB status
```

L'instruction RST &18 en &A000 exécute un saut à un sous-programme en adrobj (V), status déterminant si c'est la ROM ou la RAM qui est sélectionnée. Pour status, on utilise les conventions suivantes :

Status	! &0-&3FFF (système d'exploitation)	! &C000-&FFFF(BASIC)
&FC =252: !	ROM	! ROM
&FD =253: !	RAM	! ROM
&FE =254: !	ROM	! RAM
&FA =255: !	RAM	! RAM

Toutes les autres valeurs de status sélectionnent une ROM d'extension.

La zone de &4000 à &BFFF est une zone d'adresses en RAM.

Le nom de Far Call (appel de loin) marque bien qu'à travers l'instruction RST &18 des sauts dans toutes les RAMs et ROMs de l'ordinateur sont possibles. Le Far Call a le même effet qu'un CALL, c'est-à-dire que l'exécution du programme après l'instruction RET se poursuit à partir de l'instruction suivant l'instruction d'appel RST &18.

Si nous voulons donc lire la ROM avec la routine de moniteur, nous devons l'appeler à travers l'instruction RST &18. L'adresse indiquée par le vecteur de saut sera alors l'adresse de début de la routine de moniteur. Pour sélectionner les deux ROMs, le status (qui indique la configuration choisie) doit valoir 252. L'extension du programme se présente ainsi :

```
10  'org &a000
20  'rst &18
30  'dw vector
40  'ret ;retour au Basic
50  'vector dw du monito ;adresse vecteur de saut
60  'status db 252 ;etat ROM/RAM
70  'print equ &bb5a
80  'Deb DS 2
90  'Fin DS 2
100 'monito ld hl,(Deb)
```

Voici maintenant le listing assembleur complet :

```
A000              10            ORG  &a000
A000   DF         20            RST  &18
A001   0000       30            DW   vector
A003   C9         40            RET   ;retour au Basic
**** Linge 30 : VECTOR=&A004
A004   0000       50 VECTOR     DW   monito ;adresse vecteur de saut
A006   FC         60 STATUS     DB   252 ;etat ROM/RAM
A007              70 PRINT      EQU  &bb5a
A007              80 DEB        DS   2
A009              90 FIN        DS   2
**** Ligne 50 : MONITO=&A00B
```

```
A00B  2A07A0    100 MONITO  LD    hl,(Deb)
A00E  7C        110 SUI16   LD    a,h ;sortir octet fort
A00F  CD0000    120         CALL  sorhex
A012  7D        130         LD    a,l ;sortir octet faible
A013  CD0000    140         CALL  sorhex
A016  3E20      150         LD    a,&20 ;ASCII de l'espace
A018  CD5ABB    160         CALL  print
A01B  0610      170         LD    b,16 ;compteur
A01D  7E        180 SUI     LD    a,(hl) ;charger octet dans accu
A01E  CD0000    190         CALL  sorhex ;et sortir
A021  3E20      200         LD    a,&20 ;espace
A023  CD5ABB    210         CALL  print ;sortir
A026  23        220         INC   hl
A027  10F4      230         DJNZ  sui
A029  3E20      240         LD    a,&20
A02B  CD5ABB    250         CALL  print ;espace
A02E  111000    260         LD    de,16
A031  B7        270         OR    a ;carry=0
A032  ED52      280         SBC   hl,de ;diminuer pointeur de 16
A034  0610      290         LD    b,16
A036  7E        300 SUIAS   LD    a;(hl) ;charger octet dans accu
A037  23        310         INC   hl ;augmenter pointeur
A038  CBBF      320         RES   7,a
;convertir caractere graphique en ASCII
A03A  FE20      330         CP    &20 ;superieur egal 32??
A03C  30FE      340         JR    nc,pr ;oui, alors sortie
A03E  3E2E      350         LD    a,46 ;ASCII pour point
**** Ligne 340 : PR=&A040
A040  CD5ABB    360 PR      CALL  print ;sortir caractere
A043  10F1      370         DJNZ  suias
A045  3E0D      380         LD    a,13 ;carriage return
A047  CD5ABB    390         CALL  print ;sortir
A04A  3E0A      400         LD    a,10 ;line feed
A04C  CD5ABB    410         CALL  print ;sortir
A04F  E5        420         PUSH  hl ;sauver pointeur
A050  ED5B09A0  430         LD    de,(Fin)
A054  B7        440         OR    a ;carry=0
A055  ED52      450         SBC   hl,de ;pointeur-Fin<=0
A057  E1        460         POP   hl
;amener pointeur (pas de modification des flags!!)
```

```
A058  38B4     470          JR    c,sui16 ;hl-de<0, alors suite
A05A  28B2     480          JR    z,sui16 ;hl-de=0, alors suite
A05C  C9       490          RET    ;hl-de>0, alors fin
A05D           500                 ;fin moniteur
A05D           510                 ; sortie hexa
A05D           520                 ; sort le contenu de l'accu en
                                  hexa
A05D           530                 ; le registre E est modifie
**** Ligne 120 : SORHEX=&A05D
**** Linge 140 : SORHEX=&A05D
**** Ligne 190 : SORHEX=&A05D
A05D  5F       540 SORHEX   LD    e,a ;stockage provisoire accu
A05E  0F       550          RRCA  ;decaler
A05F  0F       560          RRCA  ;de 4 bits
A060  0F       570          RRCA  ;vers la
A061  0F       580          RRCA  ;droite
A062  E60F     590          AND  &x1111 ;annuler bits 4-7
A064  CD0000   600          CALL conv ;sortir quartet superieur
A067  7B       610          LD    a,e ;ancien contenu accu
A068  E60F     620          AND  &x1111 ;annuler bits 4-7
A06A  CD0000   630          CALL conv ;sortir quartet inferieur
A06D  C9       640          RET    ;fin sortie hexa
A06E           650                 ;routine de conversion
A06E           660
;sort le chiffre hexa correspondant au contenu de l'accumulateur
**** Ligne 600 : CONV=&A06E
**** Ligne 630 : CONV=&A06E
A06E  FE0A     670 CONV     CP    &a ;valeur du chiffre<10
A070  38FE     680          JR    c,chiffr ;oui alors chiffre
A072  C607     690          ADD  a,7 ;ajouter 7 pour les lettres
**** Ligne 680 : CHIFFR=&A074
A074  C630     700 CHIFFR   ADD  a,48 ;=code ascii des chiffres
                                   hexa
A076  CD5ABB   710          CALL print
A079  C9       720          RET    ;fin conversion

programme : monitor
Debut &A000          Fin : &A079
Longueur : 007A
   O  Erreur
```

Table de variables :

VECTOR	A004	STATUS	A006	PRINT	BB5A	DEB	A007	FIN	A009
MONITO	A00B	SUI16	A00E	SUI	A01D	SUIAS	A036	PR	A040
SORHEX	A05D	CONV	A06E	CHIFFR	A074				

Le programme Basic d'entrée :

```
10 REM Programme d'appel du MONITEUR
20 MEMORY &9FFF
30 LOAD"monitor.obj"
40 r$(0)="ROM":r$(1)="RAM"
50 MODE 2
60 PRINT"M O N I T E U R   D E   L E C T U R E   R O M / R A M
70 INPUT"Adresse de debut : &",a$
80 adr=&A007:GOSUB 220
90 INPUT"Adresse de fin : &",a$
100 adr=&A009:GOSUB 220
110 p=VPOS(#0)
120 PRINT"Systeme d'exploitation :";r$(syststa);CHR$(13);
130 a$=INKEY$:IF a$="" THEN 130
140 IF a$<>CHR$(13) THEN syststa=syststa XOR 1:GOTO 120 ELSE PRINT
150 PRINT"BASIC                   :";r$(basista);CHR$(13);
160 a$=INKEY$:IF a$="" THEN 160
170 IF a$<>CHR$(13) THEN basista=basista XOR 1:GOTO 150 ELSE PRINT
180 status=&X11111100 OR basista*2 OR syststa
190 POKE &A006,status
200 CALL &A000
210 GOTO 70
220 a=VAL("&"+a$)
230 IF a<0 THEN a=a+2^16
240 ah=INT(a/256):POKE adr+1,ah
250 POKE adr,a-ah*256
260 RETURN
```

Pour sélectionner l'état (status) ROM/RAM, >ENTER< accepte l'état proposé. En appuyant sur n'importe quelle autre touche, vous pouvez modifier l'état proposé. Nous pouvons maintenant nous commencer notre voyage dans la ROM.

Lancer le programme et entrez :

Adresse de début	: &CC00 >ENTER<
Adresse de fin	: &CE00 >ENTER<
Système d'exploitation	: ROM >ENTER<
BASIC	: ROM >ENTER<

Si vous entrez les mêmes adresses, mais que vous changez l'état de la zone Basic en RAM, au lieu des messages d'erreur de l'ordinateur (qui sont bien sûr en ROM), vous obtenez uniquement un Hexdump de la mémoire écran (RAM).

Vers l'adresse $660 (ROM) figure le message de mise sous tension de l'ordinateur. En ROM Basic figure aux alentours de l'adresse &E380 la liste des mots d'instruction Basic qu'utilise l'interpréteur.

Examinez un peu le contenu de la RAM et de la ROM pour vous faire un peu une idée de leur organisation.

La plus grande partie de la ROM est constituée par des programmes. Ecrivez une routine avec l'instruction RST &18 qui transmette le contenu d'une case mémoire de la ROM à un programme Basic et intégrez cette routine dans le désassembleur, dans le sous-programme en ligne 940. Vous aurez alors la possibilité d'interpréter les programmes figurant en ROM. Pour bien comprendre les routines internes de la machine, l'idéal est de disposer d'un listing commenté de la ROM tel que vous pouvez le trouver dans la Bible du programmeur de l'Amstrad CPC.

Le Breakpoint

Nous allons maintenant vous montrer comment écrire un programme de test pour vos programmes en langage-machine. Il vous est certainement déjà arrivé de "planter" l'ordinateur sans que vous

ayez la moindre idée de la raison pour laquelle cela s'est produit. Un programme en langage-machine ne sort pas de messages d'erreur et la plupart du temps il se "plante" tout simplement lorsqu'il y a une erreur. Il n'est donc pas possible de reconstituer ce qui s'est passé jusqu'au moment où l'erreur s'est produite. Il serait donc intéressant de pouvoir interrompre le programme en n'importe quel endroit pour examiner le contenu des registres. Il est en effet possible de déceler une erreur au vu de ces informations. Nous utiliserons à cet effet l'instruction RST &30. Ce Restart n'est pas utilisé par le système d'exploitation et il est à notre disposition. Les autres instructions Restart ne doivent être utilisées en aucun cas car elles remplissent toutes certaines fonctions pour le système d'exploitation. L'instruction RST &30 a le code &F7. Là où le programme doit être interrompu, nous écrirons le code &F7. Le code d'instruction &F7 a pour effet d'interrompre le programme, d'où le nom de Breakpoint. Puis le programme à tester est lancé. S'il rencontre l'instruction RST &30, un saut au sous-programme à l'adresse &30 est exécuté. Nous écrivons une instruction JP à l'adresse &30. Cette instruction effectue un saut à la routine de sortie des registres. Le listing assembleur suivant est auto-commenté :

```
A500            10          ORG  &a500
A500            20                ;activer le Breakpoint
A500  3EC3      30          LD   a,&c3 ;code pour jp
A502  323000    40          LD   (&0030), a ;charger en &30
                                 (RST)
A505  210000    50          LD   hl,redump
;adresse de debut de registre-dump
A508  223100    60          LD   (&0031),hl ;vers &31/&32
A50B  C9        70          RET   ;activer fin
A50C            80                ;register-dump
**** Ligne 50 : REDUMP=&A50C
A50C  F5        90 REDUMP   PUSHaf ;placer registres
A50D  C5       100          PUSHbc ;sur pile
A50E  D5       110          PUSHde
A50F  E5       120          PUSHhl
A510  210000   130          LD   hl,0 ;calculer
A513  39       140          ADD  hl,sp ;contenu SP
A514  110A00   150          LD   de,10 ;originel
```

```
A517  19       160          ADD  hl,de ;et placer
A518  E5       170          PUSH hl ;sur la pile
A519  060C     180          LD   b,12 ;nombre d'octets a sortir
A51B           190               ;sortie dans l'ordre
A51B           200               ;PC AF BC DE HP SP
A51B  2B       210 PRNT     DEC  hl
A51C  7E       220          LD   a,(hl) ;retirer un octet de la
                                 pile
A51D  CD0000   230          CALL sorhex ;et sortir
A520  10F9     240          DJNZ prnt
A522  E1       250          POP  hl ;amener ancien SP et ignorer
A523  E1       260          POP  hl ;retirer les
A524  D1       270          POP  de ;autres registres
A525  C1       280          POP  bc ;de la pile
A526  F1       290          POP  af
A527  DDE1     300          POP  ix ;retirer adresse de retour
A529  C9       310          RET   ;sauter au Basic
**** Ligne 230 : SORHEX=&A52A
A52A  00       320 SORHEX  NOP   ;ajouter ici routine sorhex
End Assumed

Programme: Breakpoi
Debut : &A500        Fin : &A52A
Longueur : 002B
  0 Erreur
Table de variables :
REDUMP A50C PRNT A51B SORHEX A52A
```

Ici également, il vous faut bien sûr intégrer la routine sorhex. *CALL &A500 stocke l'adresse de début de la routine de register-dump* avec l'instruction JP à partir de l'adresse &30. L'instruction RST &30 est ainsi disponible pour tester des programmes. Essayez le programme suivant après CALL &A500 :

```
10  ' ORG &A000
20  ' LD A,1
30  ' LD BC,&0203
40  ' LD DE,&0405
50  ' LD HL,&0607
```

```
60  ' RET
70  ' END
```

Après l'assemblage, lancez le programme avec CALL &A000. Les valeurs 1 à 7 devraient normalement être chargées dans les registres. Pour le vérifier, remplaçons l'instruction RET par l'instruction RST &F7 :

```
POKE &A00B,&F7
```

Entrez maintenant CALL &A000 et vous obtiendrez la sortie suivante :

```
A00C0168020304050607BFF8
```

Les deux premiers octets correspondent au contenu du PC après l'interruption (l'interruption s'est produite à l'adresse &A00B). Ensuite vient l'accumulateur (=1). Puis le registre des flags :

```
&68=&X  0 1 1 0 1 0  0  0
        S Z   H P/V N  C
```

Ensuite viennent les registres B, C, D, E, H et L.

Les 4 derniers chiffres représentent le contenu de SP avant l'interruption.

Notez que le Breakpoint (le code &F7, soit RST &30) ne doit figurer, avec cette routine, que dans le niveau de programme le plus élevé. S'il est en effet rencontré dans un sous-programme, le retour correct au Basic ne peut se produire puisque POP IX ne retire qu'une adresse de retour de la pile. En se fondant sur le principe de cette routine, il est possible d'écrire des aides à la programmation pratiques comme par exemple un simulateur pas à pas. Les bons packages de logiciels contiennent de tels programmes de test.

Recherche de routine

Pour compléter votre collection de routines de moniteur, voici encore une routine qui recherche une séquence de caractères dans la mémoire. Si vous voulez également effectuer des recherches dans la ROM, il vous faut réaliser l'appel de la ROM à travers un Far Call, comme dans la routine de moniteur. Il faut également intégrer la routine "sorhex".

```
A000              10                ;recherche
A000              20          ORG  &a000
A000              30 PRINT    EQU  &bb5a
A000              40 DEBUT    DS   2
A002              50 FIN      DS   2
A004              60 LONGUR   DS   1
;longueur de la sequence de caracteres
A005              70 TAB1     DS   1 ;debut sequence de caracteres
A006              80 TAB2     DS   19 ;max. 20 caracteres reserves
A019              90                ;debut recherche
A019  3A05A0     100          LD   a,(tab1) ;premier element
A01C  ED5B00A0   110          LD   de,(debut) ;debut de bloc
A020  2A02A0     120          LD   hl,(fin) ;fin de bloc
A023  B7         130          OR   a ;carry=0
A024  ED52       140          SBC  hl,de ;longueur de bloc
A026  23         150          INC  hl ;charger dans BC
A027  44         160          LD   b,h
A028  4D         170          LD   c,l
A029  EB         180          EX   de,hl ;debut dans hl
A02A  EDB1       190 COMP     CPIR  ;chercher jusqu'a
A02C  CC0000     200          CALL z,found ;egalite puis en found
A02F  E0         210          RET  po
;RET si bloc a ete entierement parcouru
A030  18F8       220          JR   comp
**** Ligne 200 : FOUND=&A032
A032  F5         230 FOUND    PUSH af
A033  C5         240          PUSH bc
A034  E5         250          PUSH hl
A035  3A04A0     260          LD   a,(longur)
A038  4F         270          LD   c,a ;stocker longueur
A039  0600       280          LD   b,0 ;dans BC
```

```
A03B  0D       290          DEC  c
;puisqu'on compare a partir du second element
A03C  28FE     300          JR   z, ok
A03E  1106A0   310          LD   de,tab2 ;adresse second element
A041  1A       320 COMP1    LD   a,(de) ;comparer
A042  EDA1     330          CPI   ;prochain element
A044  13       340          INC  de ;augmenter pointeur
A045  20FE     350          JR   nz,cont
;different, a l'instruction CPIR
A047  EA41A0   360          JP   pe,comp1
;pas encore BC=0, alors continuer comparaison
**** Ligne 300 : OK=&A04A
A04A  E1       370 OK       POP  hl ;adresse de la sequence
                                 trouvee + 1
A04B  2B       380          DEC  hl
A04C  7C       390          LD   a,h ;octet fort
A04D  CD0000   400          CALL sorhex ;sortir
A050  7D       410          LD   a,l ;octet faible
A051  CD0000   420          CALL sorhex ;sortir
A054  3E20     430          LD   a,32 ;espace
A056  CD5ABB   440          CALL print ;sortir
A059  23       450          INC  hl ;restaurer ancienne valeur
A05A  C1       460          POP  bc
A05B  F1       470          POP  af
A05C  C9       480          RET   ;continuer recherche
**** Ligne 350 : CONT=&A05D
A05D  E1       490 CONT     POP  hl ;different
A05E  C1       500          POP  bc
A05F  F1       510          POP  af
A060  C9       520          RET   ;continuer la recherche
**** Ligne 400 : SORHEX=&A061
**** Ligne 420 : SORHEX=&A061
A061  00       530 SORHEX   NOP   ;integrer ici la routine
                                 sorhex
End Assumed

Programme : recherche
Debut : &A000        Fin : &A061
Longueur : 0062
  0 Erreur
```

Table de variables :

```
RPINT BB5A DEBUT  A000 FIN    A002 LONGUR A004 TAB1  A005
TAB2  A006 COMP   A02A FOUND  A032 COMP1  A041 OK    A04A
CONT  A05D SORHEX A061
```

La séquence à rechercher doit être stockée à partir de l'adresse &A005 avant d'appeler la routine avec CALL &A019. Ceci peut être réalisé par un programme Basic, ainsi que le fait de POKEr la longueur et les adresses de début et de fin.

Entrée de données

Nous avons jusqu'à présent rencontré des routines système qui permettent une sortie à partir du langage-machine. Nous allons maintenant nous intéresser à l'entrée de données. Nous avons jusqu'ici dû entrer et transmettre en Basic, avec des instructions POKE, les données variables telles que les adresses de début et de fin, ce qui est somme toute compliqué et peu pratique.

Le Basic de l'Amstrad nous permet cependant d'entrer des données avec l'instruction CALL. Cette instruction permet de transmettre jusqu'à 32 nombres sur deux octets. L'instruction CALL étendue a le format suivant :

```
CALL adresse,expression,expression
```

expression peut être un nombre 16 bits, une fonction ou une variable (dont la valeur doit bien sûr être un nombre 16 bits). Comme il est possible de transmettre jusqu'à 32 nombres, il n'est pas possible de les stocker tous dans les registres. Les nombres ainsi transmis sont placés sur la pile. L'accumulateur contient le nombre des expressions transmises. Le registre DE contient la dernière valeur fournie. L'adresse dans la pile où la figure la dernière entrée des nombres transmis est transmise à travers le registre IX. Le registre C contient l'état ROM/RAM (voir Far Call RST &18). Par défaut, cette valeur est toujours &FF; ce sont donc les RAMs qui sont sélectionnées. HL indique toujours l'adresse à laquelle se termine l'instruction CALL. Récapitulons :

Registre	Pas de transmission de nombres	Transmission de nombres
A	0	n (nombre)
F	F=&68 (Z=1)	F=&28 (Z=0 !)
B	&20	&20-n
C	&FF (status)	&FF
DE	Adresse à appeler	Dernier nombre transmis
HL	Adresse de fin de l'instruction CALL	
IX	Adresse de la pile &BFFE	Adresse dans la pile du dernier élément = &BFFE - 2 * n

Utilisons ce type d'entrée pour fournir les valeurs qui conviennent au programme de moniteur. Il faut transmettre:

L'adresse de début
L'adresse de fin
L'état ROM/RAM (Far Call)

L'appel aura donc le format suivant :

```
CALL &A000,adresse de début,adresse de fin,status
```

Les modifications à apporter au programme sont donc :

```
10 'ORG &A000
15 'CP 3 ;3 paramètres
20 'RET NZ ;non, alors fin
```

Il faut d'abord contrôler que 3 valeurs ont bien été entrées (A=3). Si ce n'est pas le cas, il y a retour au Basic :

```
25 'LD A,D
30 'OR A
35 'RET NZ
40 'LD A,E
45 'LD (status),A
```

En lignes 25, 30 et 35, on teste si D=0. Si D est différent de 0, le programme est terminé. Le status est un nombre sur un octet. Il est cependant également possible d'entrer deux nombres de deux octets. C'est pourquoi il convient donc de tester si le second octet, c'est-à-dire l'octet fort est égal à zéro. En lignes 40 et 45, le status entré est écrit à l'adresse qui convient pour l'instruction RST &18.

```
50 'LD E,(IX+2)
55 'LD D,(IX+3)
60 'LD L,(IX+4)
65 'LD H,(IX+5)
```

En lignes 50 et 55, l'adresse de fin transmise est chargée dans DE. En lignes 60 et 65, l'adresse de début est stockée dans HL.

```
70 'RST &18
75 'DW vector
80 'RET ;retour au Basic
85 'vector DW monito ;adresse de vecteur de saut
90 'status DS 1 ;état ROM/RAM
95 'fin DS 2
100 monito LD (fin),de

... suite inchangée
```

Après avoir assemblé le programme entier, vous obtenez par exemple avec

```
CALL &A000, &CC50, &CE60, 252
```

la sortie du message d'erreur de la ROM Basic.

Une autre routine système importante est celle qui permet d'entrer une touche. Après que &BB06 ait été appelé, l'ordinateur attend qu'une touche soit appuyée. La valeur de la touche enfoncée est alors renvoyée dans l'accumulateur.

La routine très simple suivante réalise une entrée à partir du clavier :

```
A000              10          ORG  &a000
A000              20 GET      EQU  &bb06
A000              30 PRINT    EQU  &bb5a
A000  CD06BB      40 ENTREE   CALL get
A003  CD5ABB      50          CALL print
A006  FE0D        60          CP   13 ;Enter???
A008  20F6        70          JR   nz,entree
A00A  3E0A        80          LD   a,10
A00C  CD5ABB      90          CALL print ;passage a la ligne
A00F  C9         100          RET
End Assumed

Programme : entree
Debut : &A000        Fin : &A00F
Longueur : 0010
  0 Erreur
Table de variables :
GET BB06 PRINT BB5A ENTREE A000
```

Remarque : Avec cette entrée, tous les caractères de commande CTRL fonctionnent, y compris donc par exemple CTRL L pour vider l'écran ou CTRL G pour le BIP.

7. Perspectives

Vous avez maintenant découvert les principales techniques de programmation et les principaux programmes d'aide à la programmation pour réaliser des programmes machine.

La programmation en assembleur est indispensable pour la solution de problèmes d'envergure. Le temps de développement pour des logiciels en langage-machine est cependant beaucoup plus élevé que pour des programmes écrits dans des langages plus évolués. C'est pourquoi de bons programmes d'aide au développement sont nécessaires pour arriver à une programmation efficace.

Nous allons décrire brièvement les possibilités offertes par de tels programmes. Un package de programmes de développement de programmes machine doit comprendre au minimum un programme d'assembleur et un programme de moniteur complet.

L'assembleur est indispensable pour développer des programmes d'envergure. Outre les pseudo-instructions que vous connaissez maintenant, de nombreux assembleurs offrent des possibilités qui rendent le développement des programmes encore plus facile. On peut citer par exemple la définition de macro-instructions, l'assemblage conditionnel et la possibilité d'accéder à des programmes ou variables externes.

Macro-instructions :

Il arrive souvent qu'une séquence donnée d'instructions figure plusieurs fois dans un programme. L'utilisation de macro-instructions permet d'éviter que vous ayez dans ce cas à entrer à nouveau chaque fois la même séquence d'instructions. En définissant une macro-instruction, vous donnez un nom à une séquence d'instructions. Vous pouvez ensuite placer dans le programme source le nom de la macro-instruction au lieu de réécrire la séquence d'instructions.

L'assembleur remplace automatiquement le nom de la macro-instruction par la séquence d'instructions correspondante. Par ailleurs l'utilisation de macro-instructions a également pour effet de rendre les programmes source plus clairs et plus courts.

Assemblage conditionnel :

Il est possible de faire assembler certaines parties du programme uniquement en fonction de telle ou telle condition. L'assemblage conditionnel vous permet d'écrire un programme source général tel qu'une gestion de fichiers et de l'adapter ensuite à chaque application particulière.

Programmes et variables externes

En programmation assembleur, il est très intéressant de pouvoir programmer de façon structurée. Cela signifie que des problèmes importants sont subdivisés en plusieurs petits et que chaque partie du programme est développée individuellement. Il arrive souvent que les mêmes sous-programmes reviennent constamment. C'est ainsi que nous avons par exemple utilisé la routine de sortie hexadécimale dans plusieurs programmes. Lorsqu'on dispose d'un assembleur puissant, il est possible de regrouper des routines et des variables souvent utilisées dans une bibliothèque de variables et de programmes. Les routines peuvent alors être désignées par leur nom dans le programme source et elles seront ensuite automatiquement chargées à partir de la cassette et de la disquette et insérées dans le programme objet.

Le programme qui réalise la fusion entre différents programmes machine est également appelé "Linker" (link=lien). A ce programme s'ajoute souvent ce qu'on appelle un relocator (relogeur) qui corrige les adresses qui sont modifiées du fait du déplacement et de l'insertion des programmes. Les programmes qui possèdent cependant également cette possibilité sont cependant déjà très complets et donc relativement chers. Mais ils ont bien sûr pour intérêt de rendre la programmation nettement plus pratique et plus rapide. D'autre part, de nombreux assembleurs disposent d'un

éditeur propre, c'est-à-dire que l'entrée des instructions assembleur n'est plus alors liée à un numéro de ligne.

Il existe encore de nombreuses aides à la programmation en dehors des assembleurs. La plupart sont en général regroupées dans un programme de moniteur. Vous avez déjà découvert les routines de base d'un moniteur. Le désassembleur est la plupart du temps intégré dans le programme de moniteur. Les possibilités de tester les programmes constituent une des caractéristiques importantes d'un moniteur. La possibilité de placer un breakpoint est la plus simple des possibilités de test. Les routines de test plus complètes sont souvent rassemblées dans ce qu'on appelle un debugger (programme de recherche des erreurs). A cet égard le programme le plus important est le simulateur pas à pas qui correspond à la fonction TRON du Basic Amstrad.

Il ne suffit toutefois pas de posséder de bons programmes d'aide à la programmation. Plus important est de se lancer réellement dans la pratique de la programmation en langage-machine. Ce livre vous a présenté les techniques de base qu'il est nécessaire de maîtriser pour programmer le Z80. Ce n'est que par la pratique que vous apprendrez réellement à programmer en langage-machine. Dans la réalisation de vos propres programmes en langage-machine, nous ne pouvons que vous souhaiter :

Bonne C H A N C E !!!

INSTRUCTIONS DE CHARGEMENT SUR 8 BITS (LD)

Registre source

Registre objet

	A	B	C	D	E	H	L	(HL)	data	(IX+d)	(IY+d)
A	7F	78	79	7A	7B	7C	7D	7E	3E data	DD 7E dis	FD 7E dis
B	47	40	41	42	43	44	45	46	06 data	DD 46 dis	FD 46 dis
C	4F	48	49	4A	4B	4C	4D	4E	0E data	DD 4E dis	FD 4E dis
D	57	50	51	52	53	54	55	56	16 data	DD 56 dis	FD 56 dis
E	5F	58	59	5A	5B	5C	5D	5E	1E data	DD 5E dis	FD 5E dis
H	67	60	61	62	63	64	65	66	26 data	DD 66 dis	FD 66 dis
L	6F	68	69	6A	6B	6C	6D	6E	2E data	DD 6E dis	FD 6E dis
(HL)	77	70	71	72	73	74	75		36		

Registre source

Registre objet

	A	B	C	D	E	H	L	(HL)	data	(IX+d)	(IY+d)
(IX+d)	DD 77 dis	DD 70 dis	DD 71 dis	DD 72 dis	DD 73 dis	DD 74 dis	DD 75 dis		DD 36 dis data		
(IY+d)	FD 77 dis	FD 70 dis	FD 71 dis	FD 72 dis	FD 73 dis	FD 74 dis	FD 75 dis		FD 36 dis data		

	A	I	R	(BC)	(DE)	(nn)
A	7F	ED 57	ED 5F	0A	1A	3A al ah
I	ED 47					
R	ED 4F					
(BC)	02					
(DE)	12					
(nn)	32 al ah					

INSTRUCTIONS DE TRANSFERT SUR 16 BITS

Instructions d'échange

Registre objet

```
     -----------
     ! nn !(nn)!
-----!----!----!   EX  AF,AF'  Code: 08
!    ! 01 ! ED !
! BC !data! 4B !   EXX         Code: D9
!    !data! al !
!    !    ! ah !   EX  DE,HL   Code: EB
!----!----!----!
!    ! 11 ! ED !   EX  (SP),HL Code: E3
! DE !data! 5B !
!    !data! al !   EX  (SP),IX Code: DD
!    !    ! ah !                     E3
!----!----!----!
!    ! 21 ! 2A !   EX  (SP),IY Code: FD
! HL !data! al !                     E3
!    !data! ah !
!    !    !    !
!----!----!----!
!    ! DD ! DD !
! IX ! 21 ! 2A !
!    !data! al !
!    !data! ah !
!----!----!----!
!    ! FD ! FD !
! IY ! 21 ! 2A !
!    !data! al !
!    !data! ah !
---------------
```

	AF	BC	DE	HL	SP	IX	IY
SP				F9		DD F9	FD F9
(nn)		ED 43 al ah	ED 53 al ah	22 al ah	ED 73 al ah	DD 22 al ah	FD 22 al ah

	AF	BC	DE	HL	IX	IY
PUSH	F5	C5	D6	E5	DD E5	FD E5
POP	F1	C1	D1	E1	DD E1	FD E1

INSTRUCTIONS DE TRANSFERT ET DE RECHERCHE

```
LDI  : ED A0
LDIR : ED B0
LDD  : ED A8
LDDR : ED B8
CPI  : ED A1
CPIR : ED B1
CPD  : ED A9
CPDR : ED B9
```

INSTRUCTIONS ARITHMETIQUES ET LOGIQUES SUR 8 BITS

Registre source

	A	B	C	D	E	H	L	(HL)	data	(IX+d)	(IY+d)
ABC	87	80	81	82	83	84	85	86	C6 data	DD 86 dis	FD 86 dis
ADC	8F	88	89	8A	8B	8C	8D	8E	CE data	DD 8E dis	FD 8E dis
SUB	97	90	91	92	93	94	95	96	D6 data	DD 96 dis	FD 96 dis
SBC	9F	98	99	9A	9B	9C	9D	9E	DE data	DD 9E dis	FD 9E dis
AND	A7	A0	A1	A2	A3	A4	A5	A6	E6 data	DD A6 dis	FD A6 dis
XOR	AF	A8	A9	AA	AB	AC	AD	AE	EE data	DD AE dis	FD AE dis
OR	B7	B0	B1	B2	B3	B4	B5	B6	F6 data	DD B6 dis	FD B6 dis

INSTRUCTIONS ARITHMETIQUES ET LOGIQUES SUR 8 BITS

Registre source

	A	B	C	D	E	H	L	(HL)	data	(IX+d)	(IY+d)
CP	BF	B8	B9	8A	BB	BC	BD	BE	FE data	DD BE dis	FD BE dis
INC	3C	04	0C	14	1C	24	2C	34		DD 34 dis	FD 34 dis
DEC	3D	05	0D	15	1D	25	2D	35		DD 35 dis	FD 35 dis

8 bits spécial :

DAA Code: 27
CPL Code: 2F
NEG Code: ED 44

INSTRUCTIONS ARITHMETIQUES ET LOGIQUES SUR 16 BITS

Registre objet

		BC	DE	HL	SP	IX	IY
ADD	HL	09	19	29	39		
ADD	IX	DD 09	DD 19		DD 39	DD 29	
ADD	IY	FD 09	FD 19		FD 39		FD 29
ADC	HL	ED 4A	ED 5A	ED 6A	ED 7A		
SBC	HL	ED 42	ED 52	ED 62	ED 72		
INC		03	13	29	3?	DD 23	FD 23
DEC		08	18	28	38	DD 2B	FD 2B

INSTRUCTIONS DE ROTATION ET DE DECALAGE

Registres source et objet

Registre objet

	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
RLC	CB 07	CB 00	CB 01	CB 02	CB 03	CB 04	CB 05	CB 06	DD CB dis 06	FD CB dis 06
RRC	CB OF	CB 08	CB 09	CB OA	CB 0B	CB OC	CB OC	CB OE	DD CB dis OE	FD CB dis OE
RL	CB 17	CB 10	CB 11	CB 12	CB 13	CB 14	CB 15	CB 16	DD CB dis 16	FD CB dis 16
RR	CB 1F	CB 18	CB 19	CB 1A	CB 1C	CB 1C	CB 1E	CB 1C	CB dis 1E	CB dis 1E
SLA	CB 27	CB 20	CB 21	CB 22	CB 23	CB 24	CB 25	CB 26	DD CB dis 26	FD CB dis 26
SRA	CB 2F	CB 28	CB 29	CB 2A	CB 2B	CB 2C	CB 2D	CB 2E	DD CB dis 2E	FD CB dis 2E

Registres source et objet

	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
SRL	CB 3F	CB 38	CB 39	CB 3A	CB 3B	CB 3C	CB 3D	CB 3E	DD CB dis 3E	FD CB dis 3E

RLD Code: ED 6F

RRD Code: ED 67

INSTRUCTIONS BITS

Registres source et objet

	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
0	CB 47	CB 40	CB 41	CB 42	CB 43	CB 44	CB 45	CB 46	DD CB dis 46	FD CB dis 46
1	CB 4F	CB 48	CB 49	CB 4A	CB 4B	CB 4C	CB 4D	CB 4E	DD CB dis 4E	FD CB dis 4E
2	CB 57	CB 50	CB 51	CB 52	CB 53	CB 54	CB 55	CB 56	DD CB dis 56	FD CB dis 56
3	CB 5F	CB 58	CB 59	CB 5A	CB 5B	CB 5C	CB 5D	CB 5E	DD CB dis 5E	FD CB dis 5E
4	CB 67	CB 60	CB 61	CB 62	CB 63	CB 64	CB 65	CB 66	DD CB dis 66	FD CB dis 66
5	CB 6F	CB 68	CB 69	CB 6A	CB 6B	CB 6C	CB 6D	CB 6E	DD CB dis 6E	FD CB dis 6E

INSTRUCTIONS BITS

Registres source et objet

	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
6	CB 77	CB 70	CB 71	CB 72	CB 73	CB 74	CB 75	CB 76	DD CB dis 76	FD CB dis 76
7	CB 7F	CB 78	CB 79	CB 7A	CB 7B	CB 7C	CB 7D	CB 7E	DD CB dis 7E	FD CB dis 7E

INSTRUCTIONS RES

Registres source et objet

	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d
0	CB 87	CB 80	CB 81	CB 82	CB 83	CB 84	CB 85	CB 86	DD CB dis 86	FD CB dis 86
1	CB 8F	CB 88	CB 89	CB 8A	CB 8B	CB 8C	CB 8D	CB 8E	DD CB dis 8E	FD CB dis 8E
2	CB 97	CB 90	CB 91	CB 92	CB 93	CB 94	CB 95	CB 96	DD CB dis 96	FD CB dis 96

Registres source et objet

	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
3	CB 9F	CB 98	CB 99	CB 9A	CB 9B	CB 9C	CB 9D	CB 9E	DD CB dis 9E	FD CB dis 9E
4	CB A7	CB A0	CB A1	CB A2	CB A3	CB A4	CB A5	CB A6	DD CB dis A6	FD CB dis A6
5	CB AF	CB A8	CB A9	CB AA	CB AB	CB AC	CB AD	CB AE	DD CB dis AE	FD CB dis AE
6	CB B7	CB B0	CB B1	CB B2	CB B3	CB B4	CB B5	CB B6	DD CB dis B6	FD CB dis B6
7	CB BF	CB B8	CB B9	CB BA	CB BB	CB BC	CB BD	CB BE	DD CB dis BE	FD CB dis BE

INSTRUCTIONS SET

Registres source et objet

	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
0	CB C7	CB C0	CB C1	CB C2	CB C3	CB C4	CB C5	CB C6	DD CB dis C6	FD CB dis C6
1	CB CF	CB C8	CB C9	CB CA	CB CB	CB CC	CB CD	CB CE	DD CB dis CE	FD CB dis CE
2	CB D7	CB D0	CB D1	CB D2	CB D3	CB D4	CB D5	CB D6	DD CB dis D6	FD CB dis D6
3	CB DF	CB D8	CB D9	CB DA	CB DB	CB DC	CB DD	CB DE	DD CB dis DE	FD CB dis DE
4	CB E7	CB E0	CB E1	CB E2	CB E3	CB E4	CB E5	CB E6	DD CB dis E6	FD CB dis E6
5	CB EF	CB E8	CB E9	CB EA	CB EB	CB EC	CB ED	CB EE	DD CB dis EE	FD CB dis EE

INSTRUCTIONS SET

Registres source et objet

	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)
6	CB F7	CB F0	CB F1	CB F2	CB F3	CB F4	CB F5	CB F6	DD CB dis F6	FD CB dis F6
7	CB FF	CB F8	CB F9	CB FA	CB F8	CB FC	CB FD	CB FE	DD CB dis FE	FD CB dis FE

INSTRUCTIONS DE SAUT

Condition

			C	NC	Z	NZ	PE	PO	M	N
JP	adr	C3 adr adr	DA adr adr	D2 adr adr	CA adr adr	C2 adr adr	EA adr adr	E2 adr adr	FA adr adr	F2 adr adr
JR	of	18 of-2	38 of-2	30 of-2	28 of-2	20 of-2				
CALL	adr	CD adr adr	DC adr adr	D4 adr adr	CC adr adr	C4 adr adr	EC adr adr	E4 adr adr	FC adr adr	F4 adr adr
RET		C9	D8	D0	C8	C0	E8	E0	F8	F0

```
JP (HL)    Code:E9
JP (IX)    Code:DD E9
JP (IY)    Code:FD E9
DJNZ of    Code:10 of-2
RETI       Code:ED 4D
RETN       Code:ED 45
```

Instructions Restart

	&00	&08	&10	&18	&20	&28	&30	&38
RST	C7	CF	D7	DF	E7	EF	F7	FF

INSTRUCTIONS D'ENTREE/SORTIE

	(n)	(C)
A	DB	ED 78
B		ED 40
C		ED 48
D		ED 50
E		ED 40
H		ED 48
L		ED 50

Instructions d'entrée de bloc

INI	Code: ED A2
INIR	Code: ED B2
IND	Code: ED AA
INDR	Code: ED BA

INSTRUCTIONS DE SORTIE DE BLOC

Registre

		A	B	C	D	E	H	L
OUT	(n)	D3						
OUT	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69

Instructions de sortie de bloc

```
OUTI    Code: ED A3
OTIR    Code: ED B3
OUTD    Code: ED AB
OTDR    Code: ED BB
```

Modification du Carry

```
CCF     Code: 3F
SCF     Code: 37
```

Instructions de commande

```
NOP     Code: 00     DI    Code: F3     IM 0    Code ED 46
HALT    Code: 76     EI    Code: FB     IM 1    Code ED 56
                                        IM 2    Code: ED 5E
```

TABLE DE CONVERSION DECIMAL - HEXADECIMAL - BINAIRE

décimal	hexa	binaire	décimal	hexa	binaire
0	&00	&X00000000	26	&1A	&X00011010
1	&01	&X00000001	27	&1B	&X00011011
2	&02	&X00000010	28	&1C	&X00011100
3	&03	&X00000011	29	&1D	&X00011101
4	&04	&X00000100	30	&1E	&X00011110
5	&05	&X00000101	31	&1F	&X00011111
6	&06	&X00000110	32	&20	&X00100000
7	&07	&X00000111	33	&21	&X00100001
8	&08	&X00001000	34	&22	&X00100010
9	&09	&X00001001	35	&23	&X00100011
10	&0A	&X00001010	36	&24	&X00100100
11	&0B	&X00001011	37	&25	&X00100101
12	&0C	&X00001100	38	&26	&X00100110
13	&0D	&X00001101	39	&27	&X00100111
14	&0E	&X00001110	40	&28	&X00101000
15	&0F	&X00001111	41	&29	&X00101001
16	&10	&X00010000	42	&2A	&X00101010
17	&11	&X00010001	43	&2B	&X00101011
18	&12	&X00010010	44	&2C	&X00101100
19	&13	&X00010011	45	&2D	&X00101101
20	&14	&X00010100	46	&2E	&X00101110
21	&15	&X00010101	47	&2F	&X00101111
22	&16	&X00010110	48	&30	&X00110000
23	&17	&X00010111	49	&31	&X00110001
24	&18	&X00011000	50	&32	&X00110010
25	&19	&X00011001	51	&33	&X00110011

TABLE DE CONVERSION DECIMAL - HEXADECIMAL - BINAIRE

décimal	hexa	binaire
52	&34	&X00110100
53	&35	&X00110101
54	&36	&X00110110
55	&37	&X00110111
56	&38	&X00111000
57	&39	&X00111001
58	&3A	&X00111010
59	&3B	&X00111011
60	&3C	&X00111100
61	&3D	&X00111101
62	&3E	&X00111110
63	&3F	&X00111111
64	&40	&X01000000
65	&41	&X01000001
66	&42	&X01000010
67	&43	&X01000011
68	&44	&X01000100
69	&45	&X01000101
70	&46	&X01000110
71	&47	&X01000111
72	&48	&X01001000
73	&49	&X01001001
74	&4A	&X01001010
75	&4B	&X01001011
76	&4C	&X01001100
77	&4D	&X01001101
78	&4E	&X01001110
79	&4F	&X01001111
80	&50	&X01010000
81	&51	&X01010001
82	&52	&X01010010
83	&53	&X01010011
84	&54	&X01010100
85	&55	&X01010101
86	&56	&X01010110
87	&57	&X01010111
88	&58	&X01011000
89	&59	&X01011001
90	&5A	&X01011010
91	&5B	&X01011011
92	&5C	&X01011100
93	&5D	&X01011101
94	&5E	&X01011110
95	&5F	&X01011111
96	&60	&X01100000
97	&61	&X01100001
98	&62	&X01100010
99	&63	&X01100011
100	&64	&X01100100
101	&65	&X01100101
102	&66	&X01100110
103	&67	&X01100111

TABLE DE CONVERSION DECIMAL - HEXADECIMAL - BINAIRE

décimal	hexa	binaire	décimal	hexa	binaire
104	&68	&X01101000	130	&82	&X10000010
105	&69	&X01101001	131	&83	&X10000011
106	&6A	&X01101010	132	&84	&X10000100
107	&6B	&X01101011	133	&85	&X10000101
108	&6C	&X01101100	134	&86	&X10000110
109	&6D	&X01101101	135	&87	&X10000111
110	&6E	&X01101110	136	&88	&X10001000
111	&6F	&X01101111	137	&89	&X10001001
112	&70	&X01110000	138	&8A	&X10001010
113	&71	&X01110001	139	&8B	&X10001011
114	&72	&X01110010	140	&8C	&X10001100
115	&73	&X01110011	141	&8D	&X10001101
116	&74	&X01110100	142	&8E	&X10001110
117	&75	&X01110101	143	&8F	&X10001111
118	&76	&X01110110	144	&90	&X10010000
119	&77	&X01110111	145	&91	&X10010001
120	&78	&X01111000	146	&92	&X10010010
121	&79	&X01111001	147	&93	&X10010011
122	&7A	&X01111010	148	&94	&X10010100
123	&7B	&X01111011	149	&95	&X10010101
124	&7C	&X01111100	150	&96	&X10010110
125	&7D	&X01111101	151	&97	&X10010111
126	&7E	&X01111110	152	&98	&X10011000
127	&7F	&X01111111	153	&99	&X10011001
128	&80	&X10000000	154	&9A	&X10011010
129	&81	&X10000001	155	&9B	&X10011011

TABLE DE CONVERSION DECIMAL - HEXADECIMAL - BINAIRE

décimal	hexa	binaire	décimal	hexa	binaire
156	&9C	&X10011100	182	&B6	&X10110110
157	&9D	&X10011101	183	&B7	&X10110111
158	&9E	&X10011110	184	&B8	&X10111000
159	&9F	&X10011111	185	&B9	&X10111001
160	&A0	&X10100000	186	&BA	&X10111010
161	&A1	&X10100001	187	&BB	&X10111011
162	&A2	&X10100010	188	&BC	&X10111100
163	&A3	&X10100011	189	&BD	&X10111101
164	&A4	&X10100100	190	&BE	&X10111110
165	&A5	&X10100101	191	&BF	&X10111111
166	&A6	&X10100110	192	&C0	&X11000000
167	&A7	&X10100111	193	&C1	&X11000001
168	&A8	&X10101000	194	&C2	&X11000010
169	&A9	&X10101001	195	&C3	&X11000011
170	&AA	&X10101010	196	&C4	&X11000100
171	&AB	&X10101011	197	&C5	&X11000101
172	&AC	&X10101100	198	&C6	&X11000110
173	&AD	&X10101101	199	&C7	&X11000111
174	&AE	&X10101110	200	&C8	&X11001000
175	&AF	&X10101111	201	&C9	&X11001001
176	&B0	&X10110000	202	&CA	&X11001010
177	&B1	&X10110001	203	&CB	&X11001011
178	&B2	&X10110010	204	&CC	&X11001100
179	&B3	&X10110011	205	&CD	&X11001101
180	&B4	&X10110100	206	&CE	&X11001110
181	&B5	&X10110101	207	&CF	&X11001111

TABLE DE CONVERSION DECIMAL - HEXADECIMAL - BINAIRE

décimal	hexa	binaire	décimal	hexa	binaire
208	&D0	&X11010000	234	&EA	&X11101010
209	&D1	&X11010001	235	&EB	&X11101011
210	&D2	&X11010010	236	&EC	&X11101100
211	&D3	&X11010011	237	&ED	&X11101101
212	&D4	&X11010100	238	&EE	&X11101110
213	&D5	&X11010101	239	&EF	&X11101111
214	&D6	&X11010110	240	&F0	&X11110000
215	&D7	&X11010111	241	&F1	&X11110001
216	&D8	&X11011000	242	&F2	&X11110010
217	&D9	&X11011001	243	&F3	&X11110011
218	&DA	&X11011010	244	&F4	&X11110100
219	&DB	&X11011011	245	&F5	&X11110101
220	&DC	&X11011100	246	&F6	&X11110110
221	&DD	&X11011101	247	&F7	&X11110111
222	&DE	&X11011110	248	&F8	&X11111000
223	&DF	&X11011111	249	&F9	&X11111001
224	&E0	&X11100000	250	&FA	&X11111010
225	&E1	&X11100001	251	&FB	&X11111011
226	&E2	&X11100010	252	&FC	&X11111100
227	&E3	&X11100011	253	&FD	&X11111101
228	&E4	&X11100100	254	&FE	&X11111110
229	&E5	&X11100101	255	&FF	&X11111111
230	&E6	&X11100110			
231	&E7	&X11100111			
232	&E8	&X11101000			
233	&E9	&X11101001			

Abréviations utilisées dans les listes d'instructions :

Instruction assembleur	! Code
adr -adresse 16 bits	! <---al---> (Adresse Low) ! <---ah---> (Adresse High)
data -données 8 bits (constante)	! <---co--->
data16 -données 16 bits (constante)	! <---cl---> ! <---ch--->
dis -distance	! <---dis-->
rpa -paire de registres BC,DE,HL,AF	! pp
rps -paire de registres BC,DE,HL,SP	! pp
reg -registres A,B,C,D,E,H,L	! rrr
req -registres source " "	! qqq
xy -pour IX ou IY	! x correspond à 0 => IX ! x correspond à 1 => IY ! (Ex: 11x11101 (DD/FD))
() -contenu de la case mémoire	!
B -numéro de bit	! bbb
of -offset/distance de saut	! of-2
cond -condition Z,NZ,C,NC,PO,PE,M,P	! ccc
con -C,NC,Z,NZ	! cc

Flags :

1 - le flag est mis après l'opération
0 - le flag est annulé après l'opération
U - le flag est indéterminé après l'opération
X - le flag est mis ou annulé suivant le résultat de l'opération
P - le flag P/V indique la parité
 - (espace) : aucune modification
! - particularité

Explication pour les tables suivantes

Dans la première table, une flèche figure pour les codes &CB, &ED, &DD et &FD. Cela a la signification suivante :

&CB: si le premier code à traduire est &CB, il faut rechercher le second code dans la deuxième table. Il s'agit des instructions de rotation et de décalage.

&ED: si le premier code à traduire est &ED, il faut rechercher le second code dans la troisième table. Il s'agit des instructions de rotation et de décalage.

&DD et &FD: si le premier code à traduire est &DD ou &FD, il s'agit d'instructions avec adressage indexé. Avec &DD, c'est le registre IX qui est concerné et avec &FD c'est le registre IY. Les instructions avec adressage indexé ne sont pas présentées dans une table spéciale. Elles peuvent être déterminées de la façon suivante à partir des tables existantes :

Il faut rechercher le second code dans les tables comme d'habitude. L'instruction obtenue doit contenir le registre HL. Si le registre HL ne figure pas dans l'opérande ou si l'instruction déterminée est EX DE,HL, c'est qu'il s'agit d'une instruction incorrecte (qui sera sortie par le désassembleur sous forme de ???). S'il s'agit d'une instruction correcte, le registre HL doit être remplacé par IX ou IY.

HL devient IX ou IY

HL devient (IX+dis) ou (IY+dis), ceci étant indiqué par le troisième code.

Ces règles s'appliquent pour toutes les instructions qui contiennent HL, à l'exception de JP (HL).

	0	1	2	3	4	5	6	7	...
0	NOP	LD BC,nn	LD (BC),A	INC BC	INC B	DEC B	LD B,n	RLCA	...
1	DJNZ of	LD DE,nn	LD (DE),A	INC DE	INC D	DEC D	LD D,n	RLA	...
2	JR NZ,of	LD HL,nn	LD (nn),HL	INC HL	INC H	DEC H	LD H,n	DAA	...
3	JR NC,of	LD SP,nn	LD (nn),A	INC SP	INC (HL)	DEC (HL)	LD (HL),n	SCF	...
4	LD B,B	LD B,C	LD B,D	LD B,E	LD B,H	LD B,L	LD B,(HL)	LD B,A	...
5	LD D,B	LD D,C	LD D,D	LD D,E	LD D,H	LD D,L	LD D,(HL)	LD D,A	...
6	LD H,B	LD H,C	LD H,D	LD H,E	LD H,H	LD H,L	LD H,(HL)	LD H,A	...
7	LD (HL),B	LD (HL),C	LD (HL),D	LD (HL),E	LD (HL),H	LD (HL),L	HALT	LD (HL),A	...
8	ADD A,B	ADD A,C	ADD A,D	ADD A,E	ADD A,H	ADD A,L	ADD A,(HL)	ADD A,A	...
9	SUB B	SUB C	SUB D	SUB E	SUB H	SUB L	SUB (HL)	SUB A	...
A	AND B	AND C	AND D	AND E	AND H	AND L	AND (HL)	AND A	...
B	OR B	OR C	OR D	OR E	OR H	OR L	OR (HL)	OR A	...
C	RET NZ	POP BC	JP NZ,nn	JP nn	CALL NZ,nn	PUSH BC	ADD A,n	RST & 00	...
D	RET NC	POP DE	JP NC,nn	OUT (n),A	CALL NC,nn	PUSH DE	SUB n	RST & 10	...
E	RET PO	POP HL	JP PO,nn	EX (SP),HL	CALL PO,nn	PUSH HL	AND n	RST & 20	...
F	RET P	POP AF	JP P,nn	DI	CALL P,nn	PUSH AF	OR n	RST & 30	...

	8	9	A	B	C	D	E	F
0	EX AF,AF'	ADD HL,BC	LD A,(BC)	DEC BC	INC C	DEC C	LD C,n	RRCA
1	JR of	ADD HL,DE	LD A,(DE)	DEC DE	INC E	DEC E	LD E,n	RRA
2	JR Z,of	ADD HL,HL	LD HL,(nn)	DEC HL	INC L	DEC L	LD L,n	CPL
3	JR C,of	ADD HL,SP	LD A,(nn)	DEC SP	INC A	DEC A	LD A,n	CCF
4	LD C,B	LD C,C	LD C,D	LD C,E	LD C,H	LD C,L	LD C,(HL)	LD C,A
5	LD E,B	LD E,C	LD E,D	LD E,E	LD E,H	LD E,L	LD E,(HL)	LD E,A
6	LD L,B	LD L,C	LD L,D	LD L,E	LD L,H	LD L,L	LD L,(HL)	LD L,A
7	LD A,B	LD A,C	LD A,D	LD A,E	LD A,H	LD A,L	LD A,(HL)	LD A,A
8	ADC A,B	ADC A,C	ADC A,D	ADC A,E	ADC A,H	ADC A,L	ADC A,(HL)	ADC A,A
9	SBC A,B	SBC A,C	SBC A,D	SBC A,E	SBC A,H	SBC A,L	SBC A,(HL)	SBC A,A
A	XOR B	XOR C	XOR D	XOR E	XOR H	XOR L	XOR (HL)	XOR A
B	CP B	CP C	CP D	CP E	CP H	CP L	CP (HL)	CP A
C	RET Z	RET	JP Z,nn	⟶	CALL Z,nn	CALL nn	ADC A,n	RST & 08
D	RET C	EXX	JP C,nn	IN A,(n)	CALL C,nn	⟶	SBC A,n	RST & 18
E	RET PE	JP (HL)	JP PE,nn	EX DE,HL	CALL PE,nn	⟶	XOR n	RST & 28
F	RET M	LD SP,HL	JP M,nn	EI	CALL M,nn	⟶	CP n	RST & 38

	0	1	2	3	4	5	6	7
0	RLC B	RLC C	RLC D	RLC E	RLC H	RLC L	RLC (HL)	RLC A
1	RL B	RL C	RL D	RL E	RL H	RL L	RL (HL)	RL A
2	SLA B	SLA C	SLA D	SLA E	SLA H	SLA L	SLA (HL)	SLA A
3								
4	BIT 0,B	BIT 0,C	BIT 0,D	BIT 0,E	BIT 0,H	BIT 0,L	BIT 0,(HL)	BIT 0,A
5	BIT 2,B	BIT 2,C	BIT 2,D	BIT 2,E	BIT 2,H	BIT 2,L	BIT 2,(HL)	BIT 2,A
6	BIT 4,B	BIT 4,C	BIT 4,D	BIT 4,E	BIT 4,H	BIT 4,L	BIT 4,(HL)	BIT 4,A
7	BIT 6,B	BIT 6,C	BIT 6,D	BIT 6,E	BIT 6,H	BIT 6,L	BIT 6,(HL)	BIT 6,A
8	RES 0,B	RES 0,C	RES 0,D	RES 0,E	RES 0,H	RES 0,L	RES 0,(HL)	RES 0,A
9	RES 2,B	RES 2,C	RES 2,D	RES 2,E	RES 2,H	RES 2,L	RES 2,(HL)	RES 2,A
A	RES 4,B	RES 4,C	RES 4,D	RES 4,E	RES 4,H	RES 4,L	RES 4,(HL)	RES 4,A
B	RES 6,B	RES 6,C	RES 6,D	RES 6,E	RES 6,H	RES 6,L	RES 6,(HL)	RES 6,A
C	SET 0,B	SET 0,C	SET 0,D	SET 0,E	SET 0,H	SET 0,L	SET 0,(HL)	SET 0,A
D	SET 2,B	SET 2,C	SET 2,D	SET 2,E	SET 2,H	SET 2,L	SET 2,(HL)	SET 2,A
E	SET 4,B	SET 4,C	SET 4,D	SET 4,E	SET 4,H	SET 4,L	SET 4,(HL)	SET 4,A
F	SET 6,B	SET 6,C	SET 6,D	SET 6,E	SET 6,H	SET 6,L	SET 6,(HL)	SET 6,A

	8	9	A	B	C	D	E	F
0	RRC B	RRC C	RRC D	RRC E	RRC H	RRC L	RRC (HL)	RRC A
1	RR B	RR C	RR D	RR E	RR H	RR L	RR (HL)	RR A
2	SRA B	SRA C	SRA D	SRA E	SRA H	SRA L	SRA (HL)	SRA A
3	SRL B	SRL C	SRL D	SRL E	SRL H	SRL L	SRL (HL)	SRL A
4	BIT 1,B	BIT 1,C	BIT 1,D	BIT 1,E	BIT 1,H	BIT 1,L	BIT 1,(HL)	BIT 1,A
5	BIT 3,B	BIT 3,C	BIT 3,D	BIT 3,E	BIT 3,H	BIT 3,L	BIT 3,(HL)	BIT 3,A
6	BIT 5,B	BIT 5,C	BIT 5,D	BIT 5,E	BIT 5,H	BIT 5,L	BIT 5,(HL)	BIT 5,A
7	BIT 7,B	BIT 7,C	BIT 7,D	BIT 7,E	BIT 7,H	BIT 7,L	BIT 7,(HL)	BIT 7,A
8	RES 1,B	RES 1,C	RES 1,D	RES 1,E	RES 1,H	RES 1,L	RES 1,(HL)	RES 1,A
9	RES 3,B	RES 3,C	RES 3,D	RES 3,E	RES 3,H	RES 3,L	RES 3,(HL)	RES 3,A
A	RES 5,B	RES 5,C	RES 5,D	RES 5,E	RES 5,H	RES 5,L	RES 5,(HL)	RES 5,A
B	RES 7,B	RES 7,C	RES 7,D	RES 7,E	RES 7,H	RES 7,L	RES 7,(HL)	RES 7,A
C	SET 1,B	SET 1,C	SET 1,D	SET 1,E	SET 1,H	SET 1,L	SET 1,(HL)	SET 1,A
D	SET 3,B	SET 3,C	SET 3,D	SET 3,E	SET 3,H	SET 3,L	SET 3,(HL)	SET 3,A
E	SET 5,B	SET 5,C	SET 5,D	SET 5,E	SET 5,H	SET 5,L	SET 5,(HL)	SET 5,A
F	SET 7,B	SET 7,C	SET 7,D	SET 7,E	SET 7,H	SET 7,L	SET 7,(HL)	SET 7,A

	0	1	2	3	4	5	6	7
4	IN B,(C)	OUT (C),B	SBC HL,BC	LD (nn),BC	NFG	RETN	IM 0	LD I,A
5	IN D,(C)	OUT (C),D	SBC HL,DE	LD (nn),DE			IM 1	LD A,I
6	IN H,(C)	OUT (C),H	SBC HL,HL	LD (nn),HL				RRD
7			SBC HL,SP	LD (nn),SP				
8								
9								
A	LDI	CPI	INI	OUTI				
	LDIR	CPIR	INIR	OTIR				

	8	9	A	B	C	D	E	F
4	IN C,(C)	OUT (C),C	ADC HL,BC	LD BC,(nn)		RETI		LD R,A
5	IN E,(C)	OUT (C),E	ADC HL,DE	LD DE,(nn)			IM 2	LD A,R
6	IN L,(C)	OUT (C),L	ADC HL,HL	LD HL,(nn)				RLD
7	IN A,(C)	OUT (C),A	ADC HL,SP	LD SP,(nn)				
8								
9								
A	LDD	CPD	IND	OUTD				
B	LDDR	CPDR	INDR	OTDR				

MODIFICATION DES FLAGS

Instruction	C	Z	P/V	S	Commentaire
ADD;ADC;SUB;SBC;CP;NEG	x	x	V	x	Instructions 8 bits
AND;OR;XOR	0	x	P	x	
INC;DEC		x	V	x	Instructions 8 bits
ADD	x				Addition 16 bits
ADC;SBC	x	x	V	x	Instructions 16 bits
RLA;RLCA;RR;RRC;SLA;SRA;SRL	x	x	P	x	
RLD;RRD		x	P	x	
DAA	!	!	P	!	
CCF	!				!: inverse flag Carry
SCF	1				
IN reg,(C)		x	P	x	
INI;IND;OUTI;OUTD		!	U	U	!:Z=0 si B=0, sinon Z=1
INIR;INDR;OUTIR;OUTDR		1	U	U	!:Z=0 si B=0, sinon Z=1
LDI;LDD			!		!:P/V=0 si BC=0, sinon P/V=1
LDIR;LDDR			0		
CPI;CPIR;CPD;CPDR		!	!	x	!:Z=1 si A=(HL), sinon Z=0 !:P/V=0 si BC=0, sinon P/V=1
LD A,I;LD A,R		x	!	x	!:P/V=IFF-status
BIT		x	U	U	

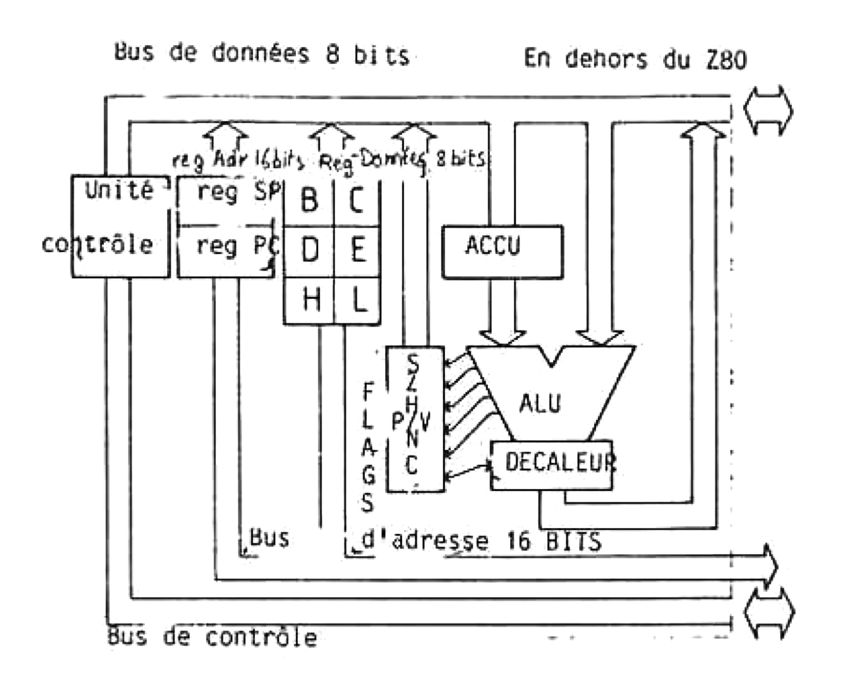


Figure 1 Structure du Z80 2.1

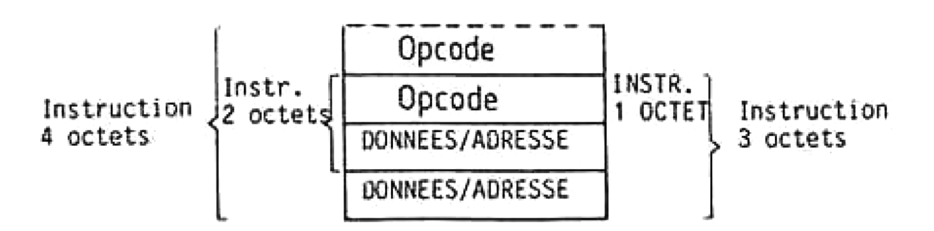
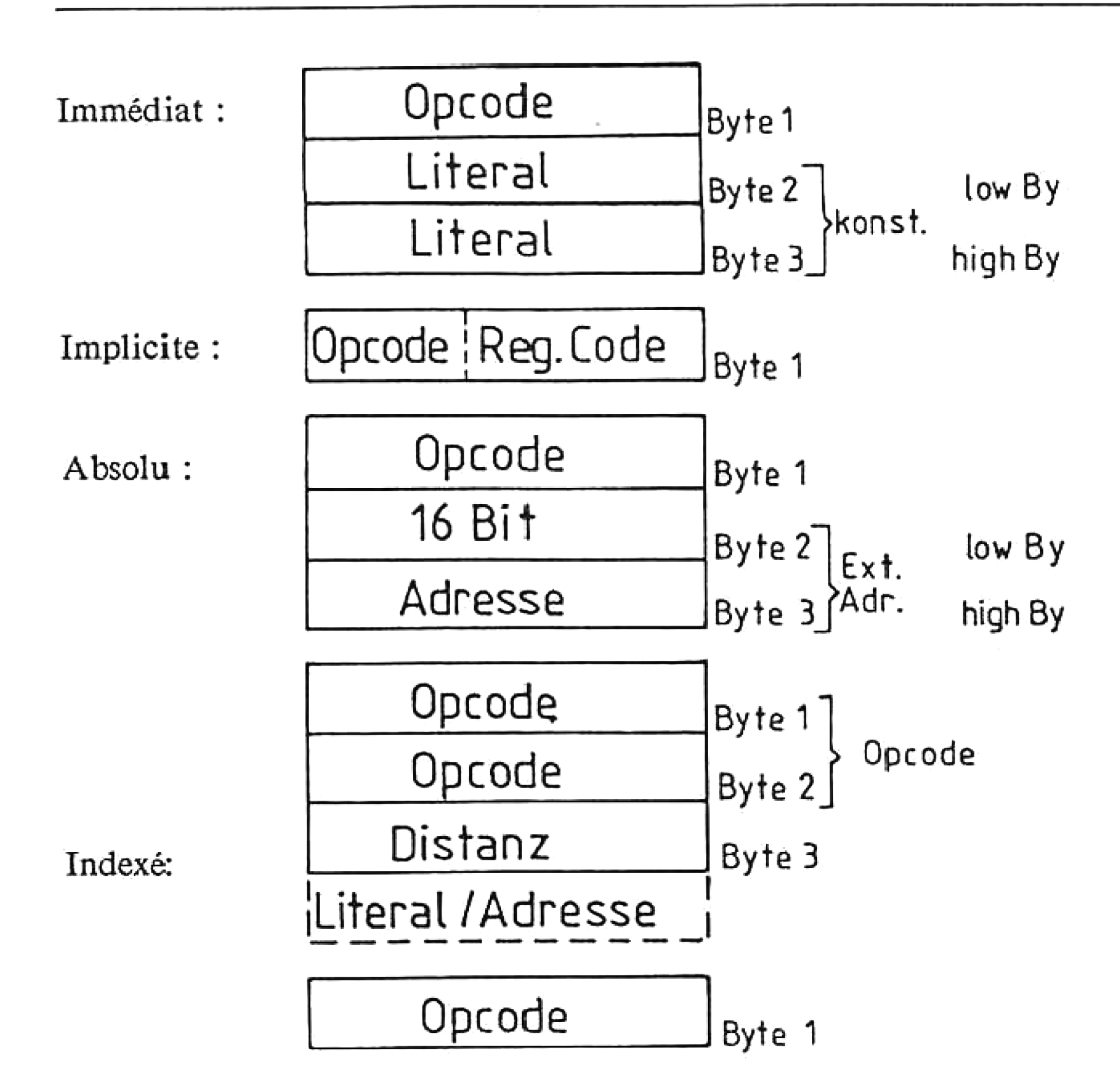

Opcode
Opcode
DONNEES/ADRESSE
DONNEES/ADRESSE
Instruction
4 octets
Instr.
2 octets
INSTR.
1 OCTET
Instruction
3 octets

Figure 2 4.1



Indirect :

Figure 3 4.1

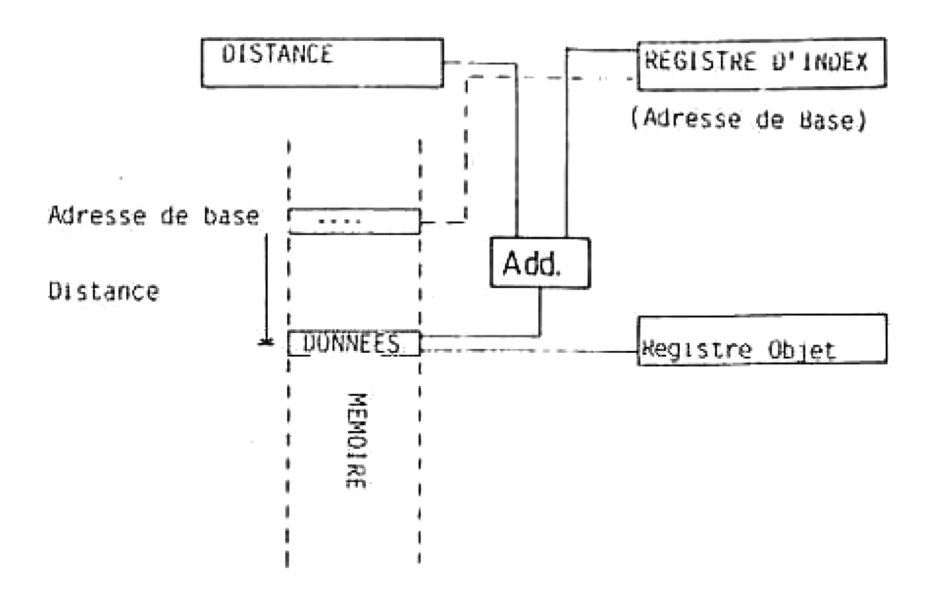


Figure 4 Adressage indexé/LD reg,(XY+dis) 4.1

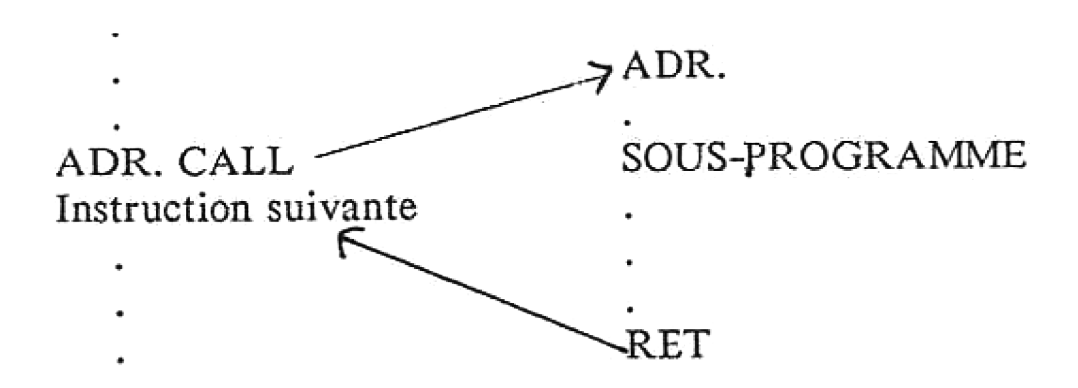


Figure 5 Appel de sous-programme 4.3

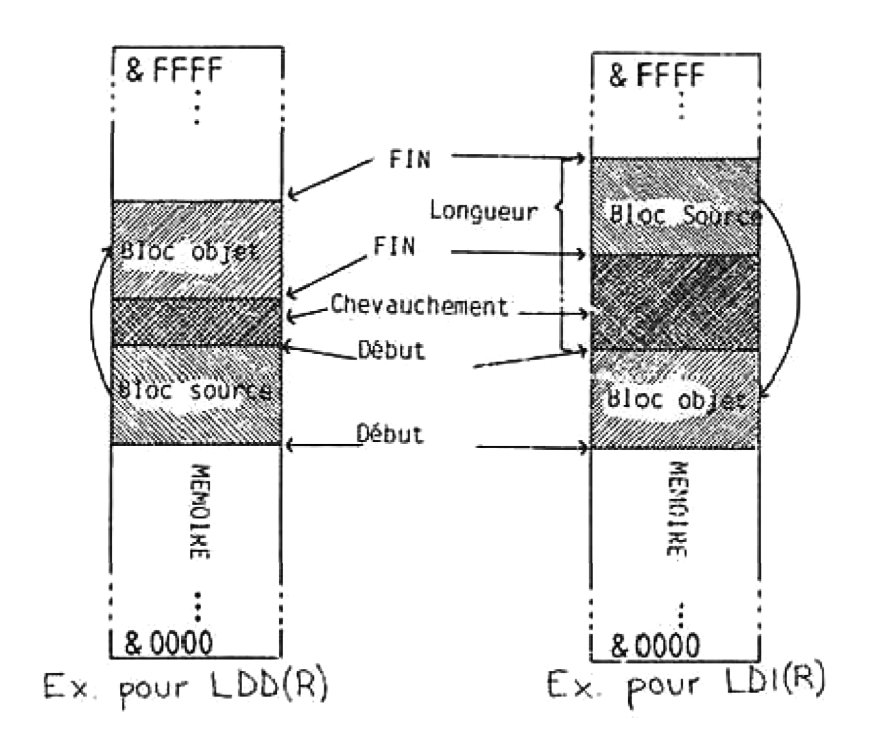
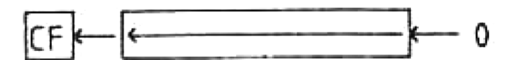


Figure 6 Instructions de transfert de bloc 4.5



SLA - décalage arithmétique vers la gauche



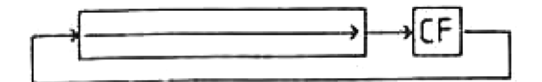
SRL - décalage logique vers la droite

Figure 7 4.8



SRA - décalage arithmétique vers la droite

Figure 8 4.8



RR - rotation vers la droite à travers le Carry

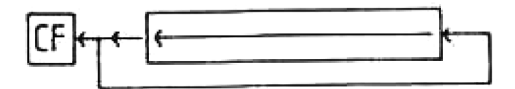


RL - rotation vers la gauche à travers le Carry

Figure 9 Rotation sur 9 bits 4.8



RRC - rotation vers la droite



RLC - rotation vers la gauche

Figure 10 Rotation sur 8 bits 4.8

Quatre octets de l'écran avant exécution



Après exécution répétée de RR

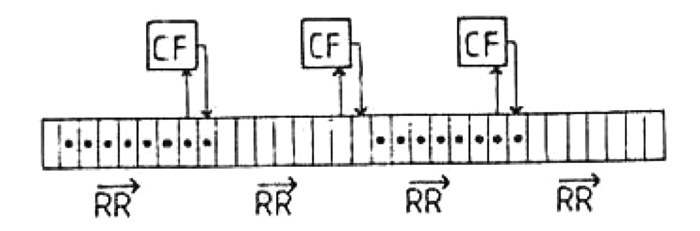


Figure 11 Application de la rotation sur 9 bits 4.8

ACHETEZ LA DISQUETTE

Ce livre vous passionne, mais vous n'avez pas le temps de taper les programmes.

Commandez nous la disquette des programmes du : livre

LE LANGAGE MACHINE POUR L'AMSTRAD CPC

BON DE COMMANDE

NOM :

PRENOM :

ADRESSE :

CODE POSTAL : VILLE :

Je désire recevoir la disquette :

LE LANGAGE MACHINE POUR L'AMSTRAD CPC

Je joins à ce coupon un [] chèque de : 120 FF [] CCP de : 120 FF

DATE : SIGNATURE :

Achevé d'imprimer en janvier 1989
sur les presses de l'imprimerie Laballery
58500 Clamecy
Dépôt légal : janvier 1989
Numéro d'impression : 901067



DULLIN/STRASSENBURG

LE LANGAGE MACHINE est l'ouvrage indispensable à tous ceux qui considèrent que le BASIC n'est ni assez puissant ni assez rapide. Des bases de la programmation en langage machine au mode de travail du processeur Z 80 avec toutes ses instructions et à l'utilisation des routines système, tout est expliqué en détail et illustré par de nombreux exemples.
Grâce à ce livre, le langage machine de votre AMSTRAD CPC n'aura plus de secret pour vous.

Principaux sujets traités :

. Structure de l'unité centrale.
. Les registres du Z 80.
. Les modes d'adressage du microprocesseur.
. Détail de chaque instruction.
. Opérations logiques et manipulation de bits.
. Un assembleur symbolique complet.
. Un désassembleur Z 80.
. Etude et réalisation d'un moniteur.
. Tableaux reprenant les instructions classées par type.

282 / ISBN : 2-86899-015-0 / ISSN : 0980-1928



Réf. ML 123 / Prix : 129 F

EDITIONS MICRO APPLICATION
58, RUE DU FAUBOURG-POISSONNIERE 75010 PARIS
TEL. (1) 47 70 32 44

7



Document restauré et retravaillé à partir d'un scan de mauvaise qualité.
AMSTRAD
CPC
MÉMOIRE ÉCRITE
ACME
https://acpc.me